UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA UNIVERSITARIA DE INGENIERÍA
TÉCNICA DE TELECOMUNICACIÓN

Proyecto Fin de Carrera

INTERACTIVIDAD PROGRAMADA EN EL GAME ENGINE DE BLENDER

Guzmán Álvarez García

Septiembre 2013

E.U.I.T. TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA
PLAN 2000

TEMA: Gráficos 3D Interactivos

TÍTULO: Interactividad programada en el Game Engine de Blender

AUTOR: Guzmán Álvarez García

TUTOR: Enrique Rendón Angulo **Vº Bº.**

DEPARTAMENTO: DIAC

Miembros del Tribunal Calificador:

PRESIDENTE: Mª Pilar Ochoa Pérez

VOCAL: Enrique Rendón Angulo

VOCAL SECRETARIO: José Luis Rodríguez Vázquez

DIRECTOR:

Fecha de lectura: 27/09/2013

Calificación: **El Secretario,**

RESUMEN DEL PROYECTO:

Este proyecto consiste en una guía introductoria al control del motor de videojuegos de la aplicación de diseño gráfico tridimensional "Blender" mediante código escrito en el lenguaje de programación "Python".

Se ha organizado en una parte teórica, centrada en el lenguaje Python y en las bases de su uso para programar el Game Engine de Blender, y otra práctica, donde se estudian de manera progresiva tres ejemplos concretos de uso del mismo: cómo controlar el objeto protagonista, cómo mover la cámara, y cómo implementar un mini-mapa de orientación. Asimismo, para complementar las explicaciones se han creado una serie de ficheros de ejemplo, todos basados en un mismo escenario (diseñado para explicar esta parte práctica) pero adaptados al concepto que se estudia en cada apartado.

Los tres ejemplos prácticos se explican exhaustivamente, y se llevan hasta un nivel relativamente alto. Se han intentado minimizar las dependencias, tanto entre ellos como con la escena que se ha usado como ejemplo, de manera que sea sencillo usar los programas generados en otras aplicaciones. Además, la mayoría de los problemas que ha sido necesario resolver durante el desarrollo no son específicos de ninguna de las tres áreas, sino que son de carácter general, por lo que sus explicaciones podrán usarse al afrontar otras situaciones.

RESUMEN

Con este proyecto se ha desarrollado una guía introductoria a uno de los aspectos más complejos y especializados de Blender, que es el control de su motor de videojuegos mediante programas escritos en Python. Está orientado a lectores que tienen un conocimiento amplio sobre el manejo de Blender, su interfaz y el funcionamiento de sus diferentes elementos, así como una mínima experiencia en cuanto a programación.

Se ha organizado en una parte descriptiva, centrada en el lenguaje Python y en las bases de su uso para programar el motor de videojuegos (Game Engine) de Blender, y otra de práctica guiada, que constituye la mayoría del proyecto, donde se estudian de manera progresiva ejemplos concretos de uso del mismo.

En la parte descriptiva se ha tratado tanto el funcionamiento más básico del lenguaje Python, especialmente las características que difieren de otros lenguajes de programación tradicionales, como su relación con Blender en particular, explicando las diferentes partes de la API de Blender para Python, y las posibles estrategias de uso.

La parte práctica guiada, dado que esta interacción entre Blender y Python ofrece un rango de posibilidades muy amplio, se ha centrado en tres áreas concretas que han sido investigadas en profundidad: el control del objeto protagonista, de la cámara y la implementación de un mapa de orientación. Todas ellas se han centrado en torno a un ejemplo común, que consiste en un videojuego muy básico, y que, gracias a los ficheros de Blender que acompañan a esta memoria, sirve para apoyar las explicaciones y poder probar su efecto directamente.

Por una parte, estos tres aspectos prácticos se han explicado exhaustivamente, y se han llevado hasta un nivel relativamente alto. Asimismo se han intentado minimizar las dependencias, tanto entre ellos como con la escena que se ha usado como ejemplo, de manera que sea sencillo usar los programas generados en otras aplicaciones.

Por otra, la mayoría de los problemas que ha sido necesario resolver durante el desarrollo no son específicos de ninguna de las tres áreas, sino que son de carácter general, por lo que sus explicaciones podrán usarse al afrontar otras situaciones.

ABSTRACT

This Thesis consists of an introductory guide to one of the most complex and specific parts of Blender, which is the control of its game engine by means of programs coded in Python. The dissertation is orientated towards readers who have a good knowledge of Blender, its interface and how its different systems work, as well as basic programming skills.

The document is composed of two main sections, the first one containing a description of Python's basics and its usage within Blender, and the second consisting of three practical examples of interaction between them, guided and explained step by step.

On the first section, the fundamentals of Python have been covered in the first place, focusing on the characteristics that distinguish it from other programming languages. Then, Blender's API for Python has also been introduced, explaining its different parts and the ways it can be used in.

Since the interaction between Blender and Python offers a wide range of possibilities, the practical section has been centered on three particular areas. Each one of the following sections has been deeply covered: how to control the main character object, how to control the camera, and how to implement and control a mini-map. Furthermore, a demonstrative videogame has been generated for the reader to be able to directly test the effect of what is explained in each section.

On the one hand, these three practical topics have been thoroughly explained, starting from the basis and gradually taking them to a relatively advanced level. The dependences among them, or between them and the demonstrative videogame, have been minimised so that the scripts or ideas can be easily used within other applications.

On the other hand, most of the problems that have been addressed are not exclusively related to these areas, but will most likely appear in different situations, thus enlarging the field in which this Thesis can be used.

A Enrique Rendón, que ha hecho posible este proyecto y cuya ayuda y dedicación han sido inestimables.

A mis padres, que siempre me han apoyado incondicionalmente.

A Saray, por su ayuda cuando me quedaba sin ideas, y su paciencia infinita durante estos nueve meses.

1. ÍNDICE

2. INTRODUCCIÓN

El objetivo de este proyecto es estudiar, resolver y documentar algunas de las necesidades más comunes a la hora de desarrollar aplicaciones interactivas mediante el software "Blender", centrándose especialmente en el uso de scripts programados en lenguaje Python para implementar la lógica en el Game Engine.

El proyecto se basa en el diseño de un juego en tercera persona consistente en una pelota que se mueve en un escenario simple, sobre el que se estudian tres aspectos diferenciados: el control de la pelota, el movimiento de la cámara, y la implementación de un mini-mapa. En cada uno de estos bloques intervienen multitud de sub-sistemas, funciones, procedimientos y algoritmos, extensibles a otras áreas de aplicación y que se explican detalladamente.

2.1. Estructura del Proyecto

El Proyecto se ha dividido en dos partes diferenciadas: la primera trata del lenguaje de programación "Python" y su relación con Blender, explicando los conceptos básicos necesarios para entender el resto de los apartados; en la segunda se aborda el estudio y la resolución de diversos problemas, estando los mismos agrupados en cuatro bloques:

- **Control del objeto protagonista**: se explican los fundamentos del movimiento, las particularidades a tener en cuenta, las posibilidades de implementación y las decisiones que se han tomado.
- **Control de la cámara**: se trata lo relacionado con el movimiento de la cámara para seguir al objeto protagonista y su interacción con el resto de elementos.
- **Implementación y control del mini-mapa de orientación**: se cubre cómo crear, añadir a la escena y controlar el comportamiento de un mini-mapa de orientación.
- **Aspectos comunes**: incluye las explicaciones que no se relacionan únicamente con uno de los apartados anteriores.

2.2. Materiales de apoyo

Como materiales de apoyo a las explicaciones escritas se han usado los siguientes recursos:

- Capturas de pantalla de Blender (incluidas en el texto, en el punto más conveniente, llamadas "Figura n").
- Ilustraciones generadas mediante programas de diseño gráfico (incluidas en el texto, en el punto más conveniente, llamadas "Figura n").
- Porciones de código donde se incluye el código relevante en cada parte de las explicaciones (incluidas en el texto, en el punto más conveniente, llamadas "Fragmento de código n"). Los fragmentos de código se han diseñado, por regla general, de forma que el primero presente en cada apartado parta de cero, y los

siguientes contengan el código que se debe cambiar/añadir para implementar las funcionalidades que se van explicando.

- Ficheros .blend con ejemplos (contenidos en la carpeta "demos", llamados "demo_*nombre_bloque*.blend"). En cada uno hay una serie de archivos de texto que corresponden a diferentes estados de la explicación, llamados "*nombre_bloque*_n". Dentro de estos archivos de texto se incluyen por lo general varios bloques de código que realizan la misma función y se usan para ejemplificar diferentes métodos o detalles, y que se deben activar/desactivar manualmente, encerrándolos entre comillas triples (comentando el código) o borrando las mismas, según se indique en la memoria. Cada archivo de texto, excepto el primero de cada bloque, contiene inicialmente la versión final (sin código intermedio) del anterior, y entre comillas triples todo el código que será necesario activar en las distintas etapas del apartado.

2.3. Consideraciones

En cuanto a la realización del proyecto y la implementación de los diferentes bloques hay que tener en cuenta:

- A fecha de cierre de esta memoria la última versión disponible de Blender es la 2.68a, y la de Python la 3.3.0. Todos los ficheros de ejemplo funcionan correctamente con ellas pero, debido a la rápida evolución del desarrollo de Blender, es posible que en un futuro se den problemas de incompatibilidad, que habrá que corregir.
- Se parte de la base de que el lector tiene conocimientos tanto de Blender como de algún lenguaje de programación. El proyecto no consiste en una guía de introducción a Blender o Python por separado, si no en la explicación y el desarrollo de ciertas formas de interacción entre ambos.
- A la hora de modelar el escenario de los ficheros de ejemplo, se ha hecho teniendo en cuenta exclusivamente las características y elementos necesarios para implementar y explicar dicha interacción, dando una importancia mínima al aspecto visual.
- Al escribir código el énfasis se ha puesto en la inteligibilidad en lugar del rendimiento. Algunos bucles o estructuras de control pueden implementarse de manera que sean más eficientes pero más difíciles de comprender, por lo que se ha decidido no hacerlo.
- En los *scripts* presentes en los archivos .blend no se han usado acentos y otros caracteres especiales deliberadamente, para evitar posibles problemas.
- Con este proyecto se busca ayudar a comprender el control del Game Engine de Blender mediante ejemplos prácticos. Para cada problema de los presentados hay un gran número de distintos caminos a seguir, de los que se ha elegido el que se ha considerado más claro y fácil. Por tanto lo escrito aquí no debe considerarse como lo único verdadero, si no como una ayuda para que cada persona resuelva los problemas como le sea más cómodo o conveniente.

- Todos los ficheros .blend de ejemplo están basados en el mismo juego, y tienen las mismas funcionalidades. Sin embargo, paralelamente la redacción de cada apartado de esta memoria se ha vuelto a implementar paso a paso el módulo correspondiente, haciendo cambios y mejoras cuando ha sido necesario. Estos cambios no siempre aparecen en el módulo correspondiente de otros ficheros de ejemplo, por tanto es importante usar el fichero correcto para seguir cada apartado, ya que contendrá la versión más avanzada de lo que en ellos se trata (además de los archivos de texto correspondientes). Por esta misma razón es posible que haya diferencias visuales o de comportamiento entre ellos, irrelevantes para el entendimiento de cada apartado.

3. FUNDAMENTOS DE PYTHON. USO DEL MISMO EN EL PROYECTO.

En este apartado se tratan diversos aspectos relacionados con el lenguaje de programación Python y con uso para interactuar con Blender, recalcando especialmente las decisiones que se han tomado en los casos donde hay más de una forma de abordar un problema.

3.1. Introducción a Python

Python es un lenguaje de programación de alto nivel que su creador, Guido van Rossum, empezó a desarrollar a finales de los 80. Puede ser usado tanto en *scripts* como en programas independientes. El aspecto que más enfatiza es la legibilidad, usando una sintaxis clara e intuitiva; en el documento "El Zen de Python" ("The Zen of Python", ver la bibliografía) se encuentran los aforismos que definen su diseño, entre los que se encuentran:

- La legibilidad cuenta.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.

Podría decirse que el lenguaje de programación al que más se parece es Java. Consideraciones generales a tener en cuenta son:

- Los bloques se diferencian mediante sangrado en lugar de paréntesis o corchetes. Por ejemplo, las acciones comprendidas dentro de un bucle "for" deben estar escritas con un número común de espacios al principio. Por comodidad suele utilizarse el tabulador, que en Blender equivale por defecto a 4 espacios (es configurable). Además cada bloque comienza con el carácter "dos puntos" (:) al final del anterior.
- No es necesario declarar las variables.
- Las condiciones de las expresiones de control (por ejemplo "if", "while") no van entre paréntesis.
- No es necesario terminar las expresiones con el carácter "punto y coma" (;).

En cuanto a los aspectos de sintaxis, Python tiene sus propias palabras clave (para expresiones de control, operadores lógicos, etc.), que en general son muy intuitivas en cuanto al significado y a su uso. En este proyecto, si bien se ofrece una visión general de Python, no se va a alcanzar ese nivel de detalle, por lo que en caso de no estar familiarizado con el lenguaje es aconsejable leer la documentación o seguir algún tutorial (ver la bibliografía).

En el Fragmento de código 1 aparece un ejemplo básico donde una función recibe una cadena de caracteres y una lista (en este caso de números), hace operaciones y escribe el resultado en pantalla. En la parte inferior se muestra una posible llamada a la función y el resultado obtenido.

```
def imprimir_media(nombre,notas):
    total = 0
    if len(notas) > 0:
        for i in range(len(notas)):
            total += notas[i] #total = total + notas[i]
        media = total / len(notas)
        print('El alumno',nombre,'tiene una calificacion media
de',media)
    else:
        print('Alumno sin calificaciones')


>>> imprimir_media('Pablo',[1,2,3,4,5,6,7,8,9,10])
El alumno Pablo tiene una calificacion media de 5.5
```

Fragmento de código 1. Ejemplo básico de construcción y llamada a una función

3.2. Objetos, tipos, clases

Según la documentación, "todos los datos en un programa escrito en Python se representan mediante objetos o relaciones entre ellos". Por tanto puede decirse que en Python todo son objetos. Cada objeto tiene asociados un identificador, único para cada uno, un tipo, que determina sus atributos y métodos (las operaciones que se pueden realizar sobre el mismo), y un valor. El identificador y el tipo de un objeto son inmutables, y el valor puede serlo o no según el tipo.

En cuanto a las clases, en Python son completamente equivalentes a los tipos, aunque al hablar de "clase" normalmente se piensa en un tipo creado por el usuario, mientras que si se dice "tipo" se suele asociar a los que existen en Python por defecto. Es común referirse a un objeto como una "instancia de una clase".

En la Tabla 1 aparecen los tipos de objetos predefinidos que más relevancia tienen en cuanto a su uso en Blender, así como otros que se usan a menudo en este proyecto:

Tabla 1. Algunos tipos de objetos usados en el proyecto

Tipo	Descripción	Ejemplo
int	Números enteros. Inmutable.	a = 1
float	Números reales. Inmutable.	b = 1.2
bool	Variable booleana. Inmutable.	True / False
str	Cadena de caracteres. Inmutable.	c = 'hola mundo'
list	Conjunto de objetos (de cualquier tipo). Mutable.	d = [a,b]
tuple	Conjunto de objetos (de cualquier tipo). Inmutable.	e = (a,b,c)
dict	Diccionario: conjunto de parejas clave-valor. Mutable.	f = {a:c,b:d}
Vector	Lista de números reales que representa un vector. Mutable.	v = Vector([1,1])
deque	Lista de objetos que permite limitar el número de elementos y añadirlos/eliminarlos por ambos extremos. Mutable.	q = deque(maxlen=8)

3.2.1. Funcionamiento de las clases

En el Fragmento de código 2 se ilustra el funcionamiento básico de las clases. Se ha implementado la misma funcionalidad que en el Fragmento de código 1, esta vez haciendo uso de la clase "alumno". Nótese la diferencia entre atributos y métodos: los atributos son propiedades del objeto, pueden ser mutables o inmutables, y se accede a ellos escribiendo el nombre del objeto, un punto y el nombre del atributo; los métodos son funciones que operan sobre los atributos, y son llamados escribiendo el nombre del objeto, un punto y el nombre del método, seguido de dos paréntesis entre los cuales se ponen los argumentos, si los hubiera.

Existe una serie de métodos especiales cuyo nombre aparece entre parejas de guiones bajos. Estos métodos son llamados por Python directamente cuando es necesario. Entre ellos probablemente el más importante es "__init__", que se conoce comúnmente como "constructor"; si está presente es llamado automáticamente al crear un objeto de esa clase, y se encarga de inicializarlo, es decir, de crear o dar valores a atributos según los parámetros pasados. Dichos parámetros son, como se aprecia en el ejemplo, los que se escriben entre paréntesis al crear los objetos de una determinada clase, que se pasan automáticamente a la función "__init__".

Puede apreciarse también cómo se tratan igualmente un tipo de objetos predefinido y uno definido por el usuario: al escribir `self.notas.append(nota)` se usa el método "append" del tipo predefinido "list", mientras que mediante `pablo.anadir_nota(9.75)` se usa el método "anadir_nota" de la clase (o tipo definido por el usuario) "alumno".

```
class alumno:
    def __init__(self,nombre,notas):
        self.nombre = nombre
        self.notas = notas

    def anadir_nota(self,nota):
        self.notas.append(nota)

    def imprimir_media(self):
        total = 0
        if len(self.notas) > 0:
            for i in range(len(self.notas)):
                total += self.notas[i] #total = total + notas[i]
            media = total / len(self.notas)
            print('El alumno',self.nombre,'tiene una calificacion
media de',media)
        else:
            print('Alumno sin calificaciones')

>>> pablo = alumno('Pablo',[6.7,5.0,9.2,7.3])
>>> pablo.imprimir_media()
El alumno Pablo tiene una calificacion media de 7.05

>>> pablo.anadir_nota(9.75)
>>> pablo.imprimir_media()
El alumno Pablo tiene una calificacion media de 7.59
```

Fragmento de código 2. Ejemplo básico de operaciones con clases, atributos y métodos

Nota: en algunas publicaciones es común encontrar la expresión “instancia de la clase X”; esto es totalmente equivalente a referirse a un objeto de la clase “X”. A todos los objetos que son de una determinada clase se les conoce como instancias de esa clase. Por ejemplo, en el Fragmento de código 2, “pablo” es una instancia de la clase “alumno”, y si se escribiese `jesus = alumno('Jesus',[7.2,8.5])`, el objeto “jesus” sería una nueva instancia de dicha clase.

3.2.2. Objetos, nombres y relaciones entre ellos

Como se ha comentado anteriormente, los objetos en Python se componen de identificador, tipo y valor (o contenido); por tanto, los objetos no tienen nombre como tal. En Python no existe el concepto de variable, entendida como en otros lenguajes de programación. En general, y en este proyecto en particular, al hablar de variables en Python se hace en realidad de nombres, cadenas de caracteres. La comprensión del sistema de nombres de Blender es de gran importancia para usar el lenguaje correctamente.

La relación entre nombres y objetos se gestiona mediante un diccionario (un objeto de tipo “dict”) conocido como “namespace”, que asocia cada nombre (cadena de caracteres) con un una referencia a un objeto. Un nombre sólo puede apuntar a un objeto, pero un objeto puede ser apuntado por varios nombres; esto, como se verá más adelante, es comúnmente causa de confusiones o problemas. Una ayuda que se sugiere frecuentemente en tutoriales sobre Python es pensar en los nombres como etiquetas que se asocian (se enganchan, se pegan) a los objetos.

A continuación se comentan varios aspectos a tener en cuenta, siguiendo el símil de las etiquetas, y en el Fragmento de código 3 se encuentran los ejemplos correspondientes.

- Un objeto puede tener asociados ninguno, uno o varios nombres. Una asignación (`cualquier_nombre = cualquier_objeto`) siempre implica colocar una etiqueta en un objeto, es decir modificar el diccionario “namespace”.
- Una asignación nunca modifica un objeto. Si un nombre existente se somete a una nueva asignación, su correspondiente entrada en el diccionario “namespace” se actualizará para contener una referencia al nuevo objeto, mientras que el anterior permanecerá invariable. Si un objeto es inaccesible (ningún nombre hace referencia a él, no tiene etiquetas) los mecanismos internos de Python lo eliminan. La primera asignación del ejemplo hace que se cree un objeto tipo “int”, de valor 1, y se le coloque la etiqueta “entero”. La segunda asignación no modifica el valor de este objeto, sino que crea otro cuyo valor es 2, y mueve la etiqueta “entero” del anterior al recién creado.
- Si en la derecha de la asignación se coloca un nombre existente, se hace que el nuevo nombre apunte al mismo objeto que el existente. En el diccionario aparecerán dos entradas, con nombres distintos pero la misma referencia: el objeto al que hacen referencia tendrá dos etiquetas. En el ejemplo, los nombres “lista” y “lista_2” apuntan a un mismo objeto, tipo “list”, con los elementos 1, 2 y 3 (al ejecutar la línea

"`lista_2 = lista`", se crea la etiqueta "lista_2" y se coloca en el mismo objeto en el que se encuentra la etiqueta "lista").

- Un nombre da acceso a los atributos y métodos del objeto al que apunta. Si un objeto está representado por varios nombres (tiene varias etiquetas), sus atributos y métodos pueden ser accedidos desde cualquiera de ellos. En el ejemplo, la línea: "`lista_2[0] = 5`" modifica el primer elemento del único objeto tipo "list" existente. Si posteriormente se usa el nombre "lista" para acceder al mismo e imprimir su contenido, se observará cómo en efecto se ha modificado su primer elemento.
- El comportamiento anterior sólo es propio de los objetos mutables (por ejemplo listas y diccionarios). Algunas operaciones que se realizan con objetos inmutables, como la del ejemplo (asignar a un número entero la suma de su propio valor y otro número) pueden dar la idea de que el valor del objeto se modifica, pero en realidad Python crea automáticamente un nuevo objeto con el resultado de la operación. En la última parte del ejemplo, al ejecutarse la línea "`dos += 1`" (equivalente a "`dos = dos + 1`", y por tanto "`dos = 1 + 1`"), Python crea un nuevo objeto de tipo entero y con valor 2, y seguidamente mueve la etiqueta "dos" del objeto anterior, de valor 1, al recién creado.
- Para realmente hacer una copia de un objeto mutable, de manera que sean independientes (lo que en otros lenguajes de programación se consigue mediante una asignación), puede usarse el método "copy", presente al menos en listas y diccionarios. Éste crea un objeto nuevo, del mismo tipo y con el mismo contenido, y devuelve su identificador, de manera que ambos objetos (el original y la copia) son equivalentes pero no el mismo. En el ejemplo se crea un nuevo objeto equivalente a la lista existente y se le asocia con el nombre (etiqueta) "lista_3"; de esta forma es posible actuar sobre él sin afectar a la lista original.
- Para comprobar si dos nombres apuntan al mismo objeto se puede usar el operador "is". Para comprobar si los objetos apuntados por dos nombres son equivalentes, es decir, su valor es el mismo, se usa el operador "==". En el ejemplo, se comprueba cómo "lista" y "lista_2" apuntan al mismo objeto, mientras que "lista_3" no lo hace. Antes de modificar el contenido del objeto apuntado por "lista_3", el operador "==" indica que es equivalente a la lista original; después de hacerlo, el operador devuelve "False".
- La modificación de objetos (aquellos que sean mutables) solamente se puede hacer utilizando los métodos que proporciona su tipo. Aunque en algunas ocasiones parezca que se modifican mediante asignaciones, en realidad Python convierte dichas asignaciones en llamadas a métodos. En el ejemplo, al hacer la asignación de la línea "`lista_2[0] = 5`", Python ejecuta en realidad el comando que aparece comentado.

```
entero = 1
# Objeto tipo "int" con la etiqueta "entero"
entero = 2
# La etiqueta "entero" se mueve a otro objeto tipo "int"

lista = [1,2,3]
# Objeto tipo "list" con la etiqueta "lista"
lista_2 = lista
# El objeto tipo "list" tiene ahora las etiquetas "lista" y "lista_2"
lista_2[0] = 5 # lista_2.__setitem__(0,5)
# Se modifica el primer elemento del objeto tipo "list".
print(lista)
# Imprime: [5, 2, 3]

lista_3 = lista.copy()
# Se crea un nuevo objeto tipo "list", con la etiqueta "lista_3"
print(lista_2 is lista)
# Imprime: True (las etiquetas "lista" y "lista_2" están en el mismo
objeto)
print(lista_3 is lista)
# Imprime: False (las etiquetas "lista" y "lista_3" no están en el
mismo objeto)
print(lista_3 == lista)
# Imprime: True (los objetos con etiquetas "lista" y "lista_3" son
equivalentes)
lista_3[0] = 10
print(lista)
# Imprime: [5, 2, 3]
print(lista_3 == lista)
# Imprime: False (los objetos con etiquetas "lista" y "lista_3" no son
equivalentes)

uno = 1
# Objeto tipo "int" con la etiqueta "uno"
dos = uno
# El mismo objeto tiene las etiquetas "uno" y "dos"
dos += 1
# Se crea un nuevo objeto con el resultado de la operación, y la
etiqueta "dos" se mueve al mismo
print(uno,dos)
# Imprime: 1 2
```

Fragmento de código 3. Ejemplos del funcionamiento del sistema de nombres en Python.

En este proyecto se usará el término “variable” para hacer alusión a las diferentes entradas del diccionario. En los casos donde se deba tener especial cuidado con la manipulación de objetos mutables, en cuanto a que al “cambiar una variable” se actúe sobre un objeto que a su vez es apuntado por otro nombre, se indicará expresamente.

3.3. API de Blender para Python

Una API (*Application Program Interface*) es un conjunto de funciones que permiten controlar ciertos servicios de un programa desde otro. La API de Blender para Python es muy extensa y en ocasiones compleja, por lo que a menudo es necesario consultar su documentación oficial (ver bibliografía) para consultar dudas. En esta API existen una serie de bibliotecas de funciones (conocidas como módulos) que hacen posible controlar

prácticamente la totalidad del programa y los datos que maneja desde Python; se clasifican en 3 tipos:

- Módulos de la aplicación (*Application Modules*): son aquellos que permiten controlar el programa (Blender) desde Python: cada botón que se ve en la interfaz gráfica, cada modificador que se aplica, en definitiva cada acción que se realiza con el teclado y ratón puede ser lanzada desde Python. Si se deja el ratón sobre cualquier elemento de la interfaz gráfica aparece en un recuadro el correspondiente comando en Python que realiza la misma acción. Asimismo, arrastrando el borde inferior de la ventana "Info" hacia abajo (ver Figura 1) aparece un área donde se muestra el código Python de todas las acciones que se llevan a cabo. En la Figura 1 aparece el resultado de, sobre la escena por defecto de Blender, pulsar la tecla "A" (se deselecciona el cubo) y seguidamente escribir en la consola de Python el comando que equivale a dicha tecla (el mismo que apareció en la parte superior), que ocasiona que se seleccionen todos los objetos. Todos los módulos de la aplicación están contenidos en un "supermódulo" llamado "bpy".

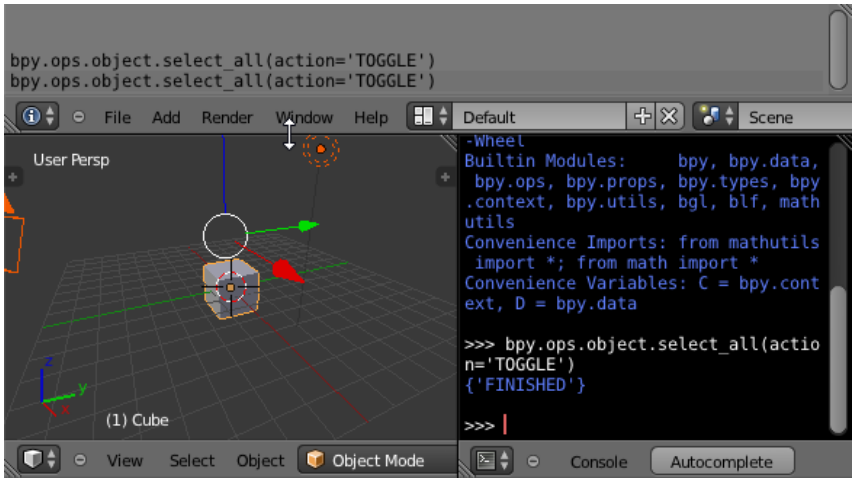


Figura 1. Resultado de seleccionar/deseleccionar todos los objetos primero mediante la tecla "A" y después mediante Python.

- Módulos independientes (*Standalone Modules*): son los que proporcionan funcionalidades que no dependen de la aplicación ni del Game Engine. Existen por ejemplo módulos dedicados a realizar funciones matemáticas ("mathutils"), o que permiten acceder al sistema de sonido ("aud").

- Módulos del Game Engine (*Blender Game Engine*): son aquellos que dan acceso a los elementos del juego cuando el mismo está corriendo. Pueden entenderse como un sustituto de los *logic bricks*, ya que son los que se usan para implementar la lógica del juego. Sin embargo aportan una flexibilidad mucho mayor ya que permiten diseñar acciones mucho más complejas y acceder a apartados como el motor de *rendering* o las texturas. Todos los módulos del Game Engine están contenidos en otro llamado "bge".

Se debe tener presente que los módulos de la aplicación no sirven para programar el Game Engine, y viceversa (los módulos del Game Engine sólo son accesibles mientras el mismo está corriendo). Como consecuencia, el botón "ejecutar script" solo tiene sentido cuando se utilizan módulos de la aplicación (por ejemplo para automatizar acciones). Este proyecto trata la programación para el Game Engine, por tanto se usarán los módulos correspondientes; sin embargo durante el desarrollo del apartado 1 (página 107), donde se explica cómo implementar un mini-mapa, se consideró necesario automatizar un proceso de copia de objetos (ver apartado 6.2.2, página 109), que sí se realiza usando los módulos de la aplicación. El código del *script* que realiza la copia se encuentra en el Anexo I.

Los módulos se importan mediante la palabra clave "import" seguida del nombre del módulo a importar. Existe la opción de importar el módulo general (bge o bpy) y acceder después a los módulos particulares, o importar directamente cada módulo particular. En este proyecto se va a utilizar el segundo método por comodidad, aunque en proyectos grandes es recomendable usar el primero para evitar conflictos de nombres. En el Fragmento de código 4 se aclara este punto: aunque en la segunda parte el módulo "logic" está correctamente importado, en caso de declarar una variable llamada "logic" se perdería el acceso a dicho módulo. En este proyecto sin embargo se usa la segunda forma, dado que al estar escrito en castellano y tener una complejidad limitada no se producirán conflictos en ese aspecto, y de esta manera el código es algo más claro.

```
# Usando el primer método
import bge
teclado = bge.logic.keyboard
logic = 'prueba' # No hay problemas en la siguiente linea, el nombre
"logic" esta libre
escena = bge.logic.getCurrentScene()

# Usando el segundo
from bge import logic
teclado = logic.keyboard
logic = 'prueba' # Despues de esta linea el objeto "logic" es una
cadena de caracteres, no un modulo
escena = logic.getCurrentScene()
# Devuelve un error:
Traceback (most recent call last):
  File "<blender_console>", line 1, in <module>
AttributeError: 'str' object has no attribute 'getCurrentScene'
```

Fragmento de código 4. Diferentes maneras de importar y usar los módulos.

Existe otra posibilidad, si bien es totalmente desaconsejable en la mayoría de los casos: importar todos los módulos particulares directamente, por ejemplo "`from bge import *`"; de esta manera no se sabe con exactitud qué nombres están ocupados, y los conflictos son muy probables. Es posible también que un *script* que funciona correctamente deje de hacerlo al cambiar de versión si se añaden módulos con nombres conflictivos. Además, los errores causados por esta práctica son en general difíciles de detectar.

Si se quieren usar módulos implementados por el usuario, por ejemplo con un conjunto de funciones no presentes por defecto que se usan en diferentes *scripts*, se deben escribir en un

fichero de texto con normalidad, importando al principio los módulos que se necesiten y definiendo funciones después, y guardarlo con extensión ".py", bien en uno de los directorios donde Python busca cuando se le ordena importar módulos o bien incluyéndolo en el propio fichero .blend. La lista de dichos directorios puede obtenerse en la consola Python de Blender, importando el módulo "sys" y seguidamente imprimiendo la variable "path": "`print(sys.path)`", aunque lo más directo es hacerlo en el mismo directorio en el que se encuentra el fichero de Blender. Si se prefiere incluirlo en el fichero basta con editarlo en el editor de texto de Blender, y se guardará automáticamente. Para importarlos se tratarán como cualquier otro módulo predefinido: `from` ***`modulo_personal`*** `import` ***`función_personal`***.

3.4. Uso de scripts en el Game Engine

Aunque los *scripts* en Python sustituyen a los *logic bricks* en cuanto a la implementación de la lógica en el Game Engine, siempre son necesarios al menos dos *logic bricks* para indicar a Blender que debe ejecutar un *script*: un sensor cualquiera conectado a un controlador tipo "python". Para determinar qué tipo de sensor es necesario conectar hay que distinguir entre los dos posibles usos que se puede dar a los *scripts*:

- Si se usan en un contexto donde la mayor parte de la lógica está implementada mediante *logic bricks*, será para realizar operaciones que requieren más complejidad que la conseguible con los *logic bricks*; en este caso el controlador "python" se activará en función de los sensores correspondientes, y generalmente lo hará en instantes puntuales. Por ejemplo, si se quisiese mostrar en pantalla cada vez que se pulsa un número su raíz cuadrada haría falta un controlador "python", que sólo funcionaría al pulsar una tecla. Un detalle a tener en cuenta es que los controladores "python" se activan tanto en el pulso positivo como en el negativo de los sensores. Este hecho toma relevancia cuando se quieren modificar variables en cantidades concretas, por ejemplo en un *script* para restar vida a un personaje. La manera de saber si el controlador se ha activado en el pulso positivo (o negativo) es comprobar si `nombre_sensor.positive == True` (ver Fragmento de código 6).
- Si, por el contrario, todo el control se realiza mediante programación (lo que se persigue en este proyecto) es necesario que el controlador "python" esté activo continuamente, de manera que en cada instante se tomen las decisiones adecuadas en función de las variables del juego; por tanto se le conectará un sensor "Always" con el modo "pulse mode" activado (ver Figura 2).

Por otra parte, e independientemente de lo anterior, el controlador tipo "Python" tiene dos modos de funcionamiento, se determinan en el campo "Execution Method" presente en el *logic brick* correspondiente (ver Figura 2):

- Modo "Script": todo el código escrito en el fichero de texto asociado al controlador se ejecuta de principio a fin en cada cuadro. Entre cuadro y cuadro todas las variables desaparecen, por tanto no se pueden usar para almacenar información del juego.

Considerando el Fragmento de código 5, si se conecta un sensor “Always” con “pulse mode” activado a un controlador “python” en modo “Script” con dichas líneas de código, en la consola se imprimirán continuamente los números del 0 al 9, una vez en cada cuadro si se comprueba que el sensor esté transmitiendo un pulso positivo, o dos si no se hace, como es el caso.

```
for i in range(10):
    print(i)
```

Fragmento de código 5. Ejemplo de un bucle “for”.

- Modo “Module”: cada controlador “python” que se configure en modo module llama a una función determinada de un fichero de texto con extensión .py. Todo el código que esté fuera de una función se ejecutará una sola vez al principio, y los módulos que se importen o las variables que se definan perdurarán hasta el final de la ejecución del juego. Dicho código se conoce como perteneciente al “top level”, en inglés, que en este proyecto se denominará “nivel superior”. Esto permite importar módulos e inicializar variables una sola vez, y almacenar datos del juego en variables, ya que éstas conservan su valor entre cuadro y cuadro. Estas ventajas que ofrece sobre el modo “Script” han llevado a que en este proyecto se haya usado este modo de funcionamiento en todos los casos. En el Fragmento de código 6 (más adelante) se ha escrito un posible uso de controladores “Python” en modo “Module”. El bucle “for” se ha incluido solamente para comparar con el ejemplo anterior: en este caso se imprimirían los números del 0 al 9 una sola vez al principio de la ejecución del juego.

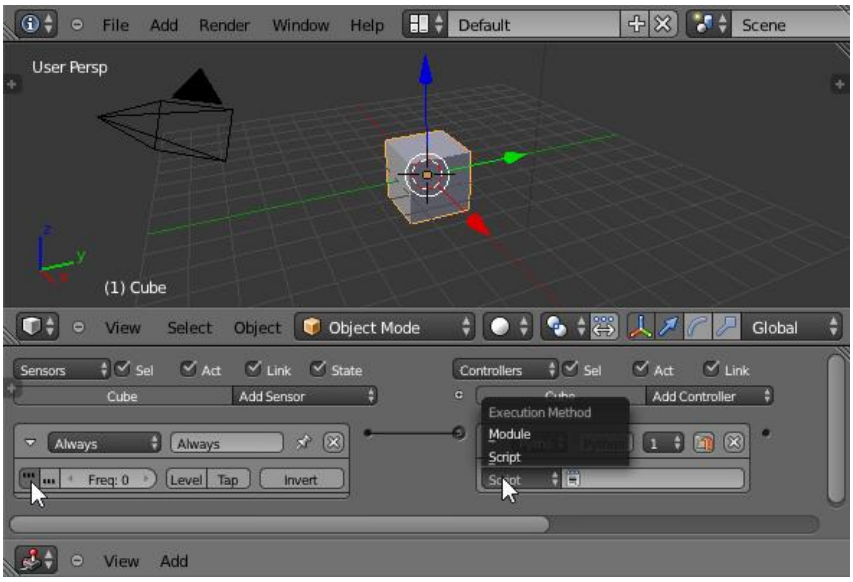


Figura 2. Posible configuración de los *logic bricks* para ejecutar un *script*.

3.5. Almacenamiento de datos

Para poder tomar las decisiones lógicas correspondientes durante la ejecución del juego es imprescindible contar con variables que almacenen datos sobre el estado de los elementos entre cuadro y cuadro.

Asumiendo que se trabaja en modo "Module", existen las siguientes posibilidades para guardar datos persistentes, es decir que se mantienen entre cuadros (los ejemplos hacen referencia al Fragmento de código 6):

- **Propiedades de objetos del juego**: todos los objetos que intervienen en el juego pueden contener propiedades, que son accesibles desde todos los *scripts*. Dentro de una propiedad se puede almacenar cualquier tipo de objeto. En el ejemplo, la cantidad de munición disponible está representada por un entero almacenado en la propiedad "municion" del objeto "arma" (`arma['municion'] = 20`). Los nombres de las propiedades de un objeto se pueden obtener mediante `nombre_objeto.getPropertyNames()`. En este proyecto las propiedades se usan para almacenar sólo aquellos datos que son compartidos entre diferentes *scripts*, ya que su uso generalizado perjudica la legibilidad.
- **Diccionario global ("globalDict")**: es un objeto de tipo "dict" que siempre está presente durante el juego, y que es único. Se accede desde cualquier *script* mediante `bge.logic.globalDict['clave']` (ver Fragmento de código 6). En dicho ejemplo se ha usado para guardar una variable que determina si el personaje está vivo. Aunque en este proyecto no ha sido necesario, se usa en general para almacenar información que debe ser consistente entre cambios, creación o re-inicialización de escenas (se pierden las referencias a los objetos), o en el resto de situaciones en las que los objetos se destruyen y por tanto se pierden las propiedades almacenadas en ellos. También se puede almacenar en un fichero para cargarlo posteriormente al principio de la ejecución, y así guardar el progreso o las preferencias del usuario.
- **Clases vacías**: es posible definir una nueva clase ("almacen_propiedades" en el ejemplo) sin atributos ni métodos, crear una instancia (llamada "propiedades" en el ejemplo), y almacenar información en atributos creados a posteriori, como se demuestra en la función "registrar_posicion" del ejemplo. Esta modalidad es válida para guardar datos que no necesitan ser accesibles desde diferentes *scripts*. Su uso se ha descartado en beneficio de las variables globales (siguiente punto), por considerarse estas últimas más intuitivas.

  Este método se basa en la posibilidad que ofrece Python de almacenar atributos en cualquier objeto. Dicha posibilidad permite consecuentemente añadir atributos a los módulos del Game Engine, escribiendo `bge.logic.`***`atributo`*** `=` ***`valor`***, lo que puede resultar ventajoso ya que, al igual que el diccionario global, los módulos del Game Engine son accesibles desde todos los *scripts* y escenas durante toda la ejecución del juego. Sin embargo se ha primado la separación entre el código presente en Python y

Blender por defecto y el introducido por el usuario: los módulos del Game Engine no se modificarán, para así evitar posibles confusiones.

- **Variables globales**: son variables de cualquier tipo que se definen en el nivel superior, y por tanto perduran entre cuadros (siempre que el controlador "Python" esté en modo "module"). Se puede acceder a ellas desde cualquier función de un mismo *script*. Un aspecto a tener en cuenta es que, dentro de las funciones, Python distingue entre variables globales y locales a la función y, por seguridad, no es posible modificar a priori las variables globales (aunque sí se puede acceder a ellas). Sirva como ejemplo la variable "vida" en el Fragmento de código 6: al estar definida en el nivel principal es global; desde cualquier función se puede consultar su valor, pero si en una de ellas se escribiese "vida = 1" se crearía una variable local llamada "vida" con el valor "1" (que se destruiría al terminar la ejecución de la función), y la variable global permanecería sin cambios. Para poder modificar el valor variable global desde una función es necesario que, en la misma función y antes de su uso, se indique expresamente, escribiendo la palabra clave "global" seguida del nombre de la variable.

  Recalcar que para definirlas, en el nivel superior, no se debe escribir primero "global"; esto sólo se hace dentro de las funciones cuando se necesita modificar el valor de una variable global desde ellas.

  Usar este método no es aconsejable si no se domina perfectamente el funcionamiento del programa, ya que puede perderse el control sobre dónde y cómo se está modificando la variable. Sin embargo en este proyecto, por tener una complejidad baja y considerarse más fáciles de manejar, se van a usar variables globales para almacenar información persistente.

En el Fragmento de código 6 se incluye una porción de lo que sería un *script* de control (llamado "juego.py") de un hipotético juego donde interviniera un personaje con un arma, implementado en modo "module". Cada función podría llamarse desde un controlador "python" conectado a los sensores correspondientes, o bien desde el mismo *script* respondiendo a otras decisiones lógicas. En la Figura 3 se muestra una posible estructura de *logic bricks* configurados para llamar a dichas funciones. Nótese cómo los sensores pertenecen a diferentes objetos (dos cubos que representan al jugador y el arma) pero están conectados a controladores que llaman a funciones del mismo *script*. La función que deben ejecutar se indica escribiendo el nombre del *script* donde se encuentra, un punto (.) y el nombre de la función, sin paréntesis (ver Figura 3).

```
###juego.py###
from bge import logic

escena = logic.getCurrentScene()
arma = escena.objects['arma']
jugador = escena.objects['jugador']
sensor_disparar = arma.sensors['disparar']
sensor_quitar_vida = jugador.sensors['quitar_vida']

class almacen_propiedades:
    pass # La clase no tiene métodos ni atributos en un principio, se
usará para almacenar información


logic.globalDict['vivo_o_muerto'] = 1    # globalDict
vida = 100                               # Variable global
arma['municion'] = 20                    # Propiedad de objeto
propiedades = almacen_propiedades()      # Instancia de clase vacía
propiedades.historial_posiciones = []    # Atributo creado

for i in range(10): # Bucle de ejemplo, solo se ejecuta una vez
    print(i)

def quitar_vida():
    global vida #Permite modificar la variable global desde la función
    if sensor_quitar_vida.positive == True and
logic.globalDict['vivo_o_muerto'] == 1:
        # Acciones correspondientes (reproducir animacion...)
        vida -= 5 # Restar 5
        if vida == 0:
            logic.globalDict['vivo_o_muerto'] = 0

def disparar():
    if sensor_disparar.positive = True:
        # Acciones correspondientes
        arma['municion'] -= 1  # Restar 1

def registrar_posicion():
    propiedades.historial_posiciones.append(jugador.position)
```

Fragmento de código 6. Ejemplo de uso de controlador Phyton en modo “module” y diferentes estrategias de almacenamiento de datos.

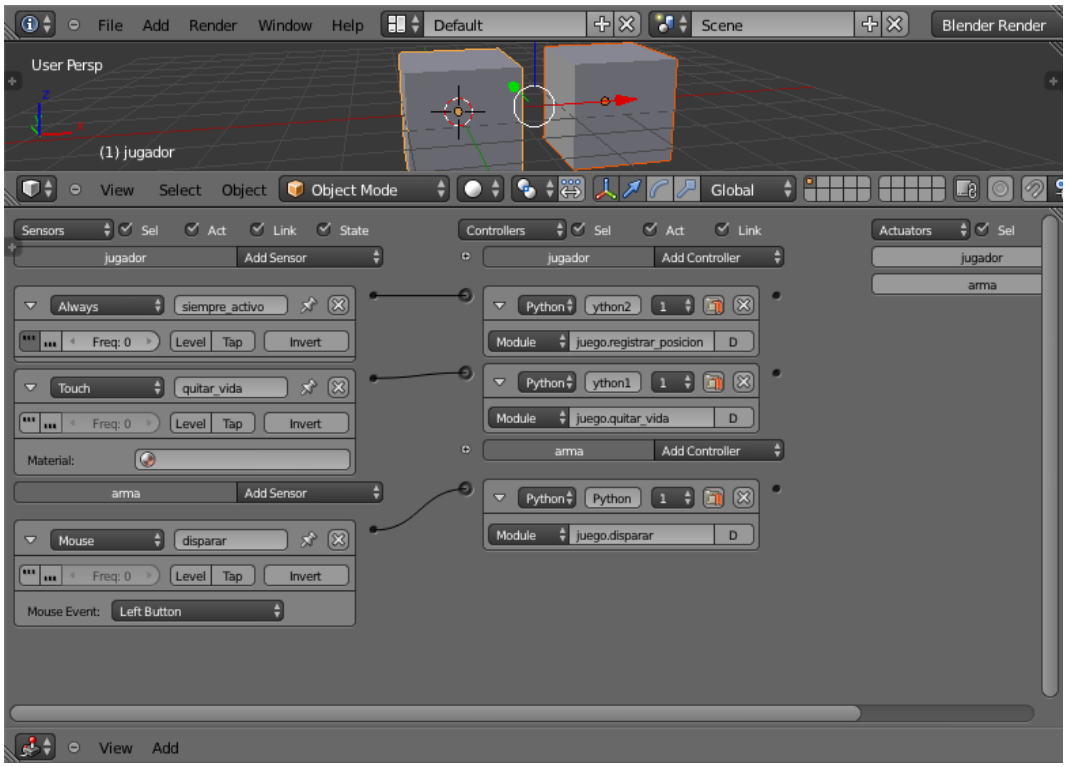


Figura 3. Posible implementación de la lógica mediante sensores y controladores "python" en modo "module".

3.6. Patrones de organización, sistema de notación

La flexibilidad que ofrecen tanto Python como Blender es tal que hay un gran número de aproximaciones distintas a la programación en el Game Engine. En este proyecto se han tomado las siguientes decisiones que se consideran útiles:

- En cada escena, colocar en el origen un objeto tipo "empty" desde el cual se lanzan todos los controladores "Python" que intervienen en dicha escena. De esta manera se separa la lógica que afecta a cada escena, y no es necesario recordar en qué objeto u objetos se colocaron los *logic bricks*, ya que siempre estarán en el *empty*.
- Agrupar en diferentes archivos de texto (módulos) las funciones que atañen a áreas determinadas. Por ejemplo, en "control_camara.py" están todas las funciones que intervienen en la lógica de la cámara, y en "control_pelota.py" las que lo hacen en la pelota.
- Llamar desde cada controlador "Python" a una sola función por cada módulo, llamada "main()", y desde ella organizar el resto de la lógica (el nombre "main" se ha elegido por mantener la similitud con otros lenguajes de programación, aunque podría usarse cualquier otro). Por ejemplo, sólo habrá un controlador "Python" correspondiente al control de la cámara, y éste lanzará "control_camara.main". Dentro del control de la cámara la lógica podría distribuirse en otras funciones, que podrían lanzarse desde controladores independientes ("control_camara.otra_función"), pero se decide hacerlo desde dicha función "main" para favorecer la inteligibilidad de la lógica.

- En cada módulo, importar en primer lugar los módulos requeridos (por ejemplo: `from math import fabs`), definir las variables globales e inicializarlas si es necesario (por ejemplo: `historial_orientacion_camara = deque(maxlen = 2)`), definir las constantes (sólo se diferencian de las anteriores en el uso que se les da; para distinguirlas se escriben con mayúsculas, por ejemplo: `FACTOR_VELOCIDAD_LATERAL = 5`), y por último codificar las diferentes funciones.
- Usar constantes tanto para establecer valores configurables por el usuario (velocidad, fuerza, etc) como para dar nombres intuitivos a constantes predefinidas en Blender. Un ejemplo de este último uso es escribir: `ACTIVO = logic.KX_INPUT_ACTIVE`. La segunda parte de la igualdad es una constante definida en el módulo "logic" de la API de Blender, que a su vez equivale al número 2; por tanto, el valor de la constante "ACTIVO" será también 2.

En cuanto a la notación, los nombres de todos los objetos, ya sean variables, módulos o funciones, se han escrito con minúsculas y usando el guión bajo (_) como separador en caso de componerse de varias palabras. Una excepción son las variables que se tratan como constantes, que se escriben con mayúsculas para facilitar su distinción.

Al elegir los nombres se ha intentado que describan lo mejor posible qué es lo que representan. Sobre este aspecto, hay que tener en cuenta que usar nombres largos en Python no afecta al rendimiento.

4. CONTROL DEL OBJETO PROTAGONISTA

4.1. Introducción

El movimiento del objeto protagonista, en este caso una pelota, se ha implementado intentando que sea simple pero se comporte con naturalidad, es decir, que responda a las órdenes o los eventos como lo haría una hipotética pelota en el mundo real.

Para conseguir dicha naturalidad, el movimiento se debe generar mediante la aplicación de fuerzas, dejando que el motor físico calcule la velocidad correspondiente según la suma de dichas fuerzas.

El hecho de implementar un videojuego en tercera persona, como se ha hecho en este proyecto, conlleva que la cámara (el punto de vista) cambia según se mueve el protagonista. Concretamente lo va siguiendo, de manera que por norma general se le ve "de espaldas" o, lo que es equivalente, la cámara tiende a apuntar en la misma dirección y el mismo sentido que el de desplazamiento del protagonista. Esto implica que las acciones que se realicen sobre la pelota deben estar siempre acorde con el punto de vista: por ejemplo, "ir hacia atrás" equivale a "desplazarse hacia la cámara", independientemente de la dirección global que lleven la cámara o la pelota.

Las opciones para el desplazamiento que se han implementado, que se explican más adelante, son las siguientes:

- Desplazamiento en el plano horizontal: giro a derecha (tecla "D") e izquierda (tecla "A"), avance hacia adelante (tecla "W") y atrás (tecla "S").
- Salto (barra espaciadora).
- Modificaciones a las anteriores, como aumentar la velocidad o parar en seco, orientadas a facilitar la programación (las pruebas) o a la implementación de "trucos".

Todas las explicaciones que se dan en este apartado se reflejan en el archivo "demo_control_pelota.blend", (excepto las del sub-apartado 4.2, que se apoyan en un fichero aparte).

4.2. Generación del movimiento

Para la función de objeto protagonista se ha elegido una esfera texturada configurada como sólido rígido ya que, si se considera sólo el control del movimiento absoluto (dejando a un lado animaciones y efectos), es el caso más complejo, dado que se basa en el rozamiento entre las superficies de la esfera y del objeto sobre el que está apoyada. Es importante comprender bien este punto, ya que es la base de todo lo que concierne al movimiento de la pelota, por tanto se va a explicar con más detalle:

La clave está en que la pelota, para que el movimiento parezca natural, tiene que rodar sobre la superficie, es decir, el punto de apoyo debe cambiar adecuadamente en cada instante (si el punto de apoyo es siempre el mismo se está produciendo un arrastre).

Por una parte es necesario aplicar una textura tanto a la superficie de la pelota como a la del objeto sobre el que apoya (en caso de que no haya otras referencias visuales). Se puede pensar en una bola de billar: si su superficie no tuviera dibujo y fuera perfecta, no sería posible determinar si se está moviendo por si misma o arrastrada por un hilo transparente; si, aunque estuviera texturada, el tapete fuese perfectamente plano y no hubiera elementos de referencia (objetos fijos como los bordes u otras bolas) tampoco podría saberse si se está moviéndose correctamente o no (podría estar elevada y por tanto girando sin ocasionar movimiento absoluto, o apoyada sobre una superficie de rozamiento cero).

Por otra parte, el movimiento se debe implementar aplicando fuerzas, tanto las que afectan a la pelota en sí como las de rozamiento. El coeficiente de rozamiento se establece mediante el valor "Friction" de los materiales, situado en la sección "Physics" de la pestaña "Material". Lógicamente hay que tener en cuenta cada uno de los materiales que intervienen y ajustar su coeficiente de rozamiento individualmente para conseguir el comportamiento deseado. Las fuerzas que se aplican a la pelota son principalmente de dos tipos ("torque" y "force"), que se ilustran en la Figura 4 y se explican a continuación:

- "Force": equivale a aplicar un empuje en una determinada dirección. Un ejemplo, acorde con la Figura 4, sería hacer rodar un neumático, en este caso empujando en la dirección negativa del eje X. De esta manera, el empuje aplicado es la causa del movimiento absoluto, y el rozamiento contra el suelo del movimiento respecto a sí misma (el giro); si el neumático se colocase sobre hielo, sería fácil moverlo sin que girase. En Python se implementa mediante `nombre_objeto.applyForce(vector, local)`, indicando mediante el parámetro "vector" la dirección y magnitud del empuje y mediante "local" los ejes que se deben tomar como referencia (los locales del objeto o los globales). Se puede decir que esta fuerza se aplica al centro de masas del objeto sin producir "torque".
- "Torque": aplica un movimiento de torsión alrededor de un determinado eje. En el mundo real se da, por ejemplo, en las ruedas motrices de los coches. En este caso el movimiento generado es el local, el giro, y el rozamiento con otra superficie produce el desplazamiento absoluto. Si se coloca un coche sobre hielo y se acelera, las ruedas motrices girarán sin que el coche se mueva apenas. Se implementa mediante `nombre_objeto.applyTorque(vector, local)`, donde el parámetro "vector" representa el eje sobre el que se realiza el giro y el parámetro "local" los ejes tomados como referencia. Dado un eje, el giro se aplica en el sentido de las agujas del reloj respecto al vector pasado. Para conseguir el movimiento representado en la Figura 4 habría que pasar el vector (0,n,0), determinando mediante "n" la fuerza con que se aplica el giro. Para mover la rueda en sentido contrario habría que cambiar el signo de "n". Se puede decir que se aplica un par de fuerzas equilibrado que sólo produce "torque".

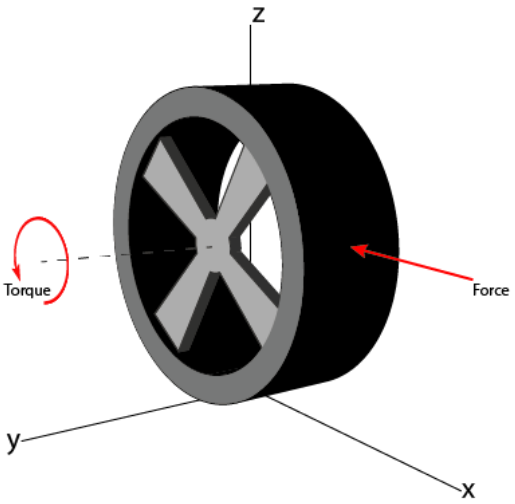


Figura 4. Esquema ilustrativo del efecto de los métodos “force” y “torque”

En el Fragmento de código 7 se presentan dos funciones y sendas llamadas a las mismas cuyo efecto sobre la rueda de la Figura 4 sería equivalente desde el punto de vista cualitativo (siempre que el rozamiento entre la rueda y el suelo sea adecuado, pues además influirían la masa y el momento de inercia de la rueda). Es conveniente usar vectores normalizados y multiplicarlos por una constante para posteriormente poder modificar la magnitud del efecto con facilidad.

```
from bge import logic

escena = logic.getCurrentScene()
rueda = escena.objects['rueda']
VELOCIDAD = 5

def avanzar_torque(magnitud):
    rueda.applyTorque(magnitud*[0,1,0],0)

def avanzar_force(magnitud):
    rueda.applyForce(magnitud*[-1,0,0],0)

>>> avanzar_torque(VELOCIDAD)
>>> avanzar_force(VELOCIDAD)
```

Fragmento de código 7. Ejemplo de uso de las funciones “applyTorque” y “applyForce”

Para demostrar el funcionamiento y la diferencia ambos métodos se ha diseñado una pequeña aplicación interactiva (fichero “demo_force_torque.blend”); mediante las teclas “t” y “f” se aplica “torque” o “force” respectivamente, mediante “x”, “y”, “z” o los números se determina el eje de actuación, y con “g” o “l” el uso de coordenadas globales o locales.

En este proyecto se ha decidido generar el movimiento más básico, el que hace moverse a la pelota sobre el suelo, aplicando “torque”. La razón para ello es que el movimiento absoluto generado depende del rozamiento entre las superficies, lo que permite distinguir entre varios

tipos de terreno (al cambiar el coeficiente de rozamiento cambia el agarre), simular “derrapes” y, lo que es más importante, hace que la pelota no se desplace (no reciba impulso adicional) al estar en el aire, eliminando la necesidad de programar lógica adicional para tenerlo en cuenta.

No obstante, se ha implementado también mediante “force”, para dar la posibilidad de controlar la pelota mientras está en el aire. Las constantes que controlan el efecto de ambos métodos son distintas, para poder configurarlos independientemente y conseguir el resultado deseado. Hay que tener en cuenta que el motor físico de Blender, llamado “Bullet”, permite controlar la masa del objeto y un factor de forma para escalar el tensor de inercia. Aparentemente, el sistema calcula o estima el tensor de inercia.

4.3. Consideraciones sobre las coordenadas.

En cuanto a la manera de controlar el objeto protagonista, el tratarse de un juego en tercera persona implica que moverlo en una misma dirección relativa (por ejemplo hacia la izquierda) equivale en la mayoría de las ocasiones a hacerlo en direcciones globales distintas. Sirva como ejemplo la Figura 5, donde se representan las situaciones de la pelota avanzando hacia adelante (sentido positivo del eje Y [0,1]) y hacia la derecha (sentido positivo del eje X [1,0]), con la cámara siguiéndola en ambos casos. En ella se observa cómo para ocasionar un giro a la izquierda en la pelota desde el punto de vista de la cámara se le debe aplicar fuerza en el sentido negativo del eje X [-1,0] en el primer caso, mientras que para hacerlo en el segundo debe aplicarse en el sentido positivo del eje Y [0,1]. Hay por tanto una diferencia importante entre usar como referencia la situación local, en la que la acción es siempre “aplicar fuerza hacia la izquierda” y la global, donde la acción es “aplicar fuerza de manera que ocasione un giro local a la izquierda”.

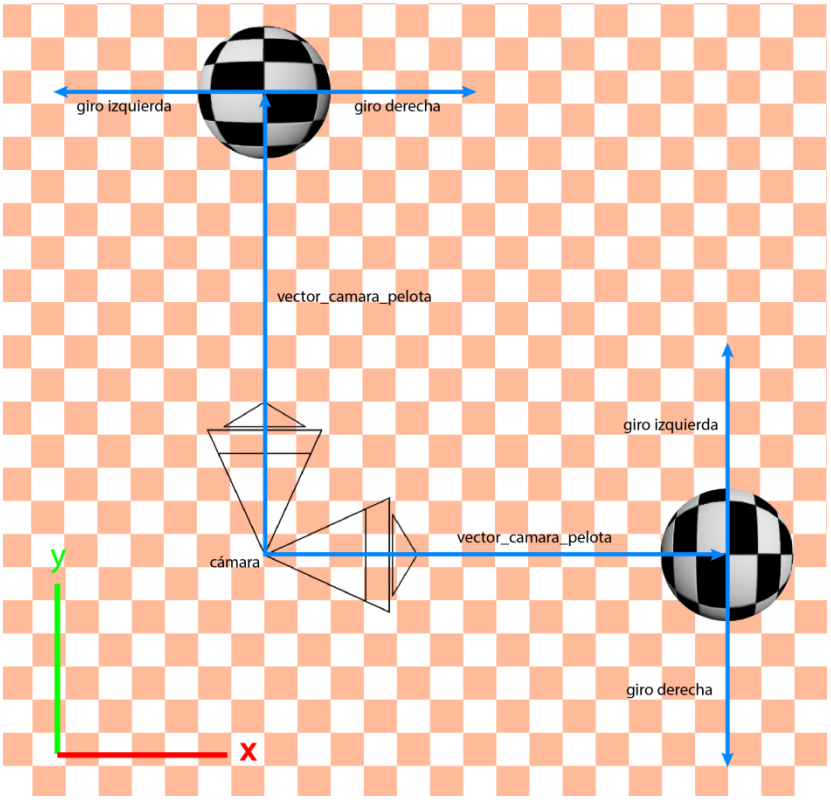


Figura 5. Esquema ilustrativo de las diferencias entre direcciones globales y locales.

En un gran número de aplicaciones este hecho no tiene mayor importancia, ya que los ejes locales del objeto protagonista tienen una orientación definida. Puede pensarse por ejemplo en el personaje Super Mario: podría establecerse que el eje Y apuntase hacia el frente, el X hacia su derecha y el Z hacia arriba y, en ese caso, avanzar siempre implicaría moverlo en el sentido positivo del eje Y (usando coordenadas locales), mientras que para hacerlo girar bastaría con aplicar una rotación alrededor del eje Z

Sin embargo, en este proyecto se ha decidido plantear el caso más complejo, que se da cuando no hay relación entre los ejes locales de un objeto y su dirección de avance: giran según lo hace la pelota, de manera que por ejemplo en un momento dado el eje Y puede estar apuntando hacia el frente, pero en otro puede apuntar hacia abajo.

Como consecuencia de lo anterior, todo el movimiento en el plano horizontal que se aplica sobre la pelota debe estar basado en su posición relativa a la cámara. Esto se consigue tomando como referencia el vector cuyo origen está en la cámara y termina en la pelota, que se ha denominado “vector_camara_pelota”.

4.4. Movimiento en el plano horizontal

En este apartado se explica cómo programar los movimientos básicos sobre el plano horizontal, es decir, hacia los lados, hacia adelante y hacia atrás. En el módulo “control_pelota_1.py”, incluido en el fichero “demo_control_pelota.blend” se encuentra el

código correspondiente. Se debe configurar el controlador "Python" del objeto "juego" para que lance el módulo y función "control_pelota_1.main".

En primer lugar se debe inicializar el controlador "Python", importando los módulos y definiendo las variables y constantes que vayan a necesitarse (ver módulo "control_pelota_1.py"). El código correspondiente se coloca en el llamado "nivel superior" (*top level*), sin estar dentro de ninguna función, de manera que se ejecute una sola vez al lanzar el juego. Inicialmente se requieren los siguientes componentes, y según se vayan añadiendo funcionalidades habrá que definir nuevas variables:

- Módulos: "logic" y "events" de "bge", "Vector" de "mathutils".
- Variables correspondientes al teclado, al ratón, a la escena y a los objetos a los que se necesita acceder (pelota y cámara).
- Constantes: las que redefinen los nombres de las teclas y los de sus estados, y las que definen otros parámetros (en este caso la velocidad).

A continuación se pasa a definir la función "main" (como se ha comentado, se ha nombrado así por comodidad, pudiendo haberse elegido prácticamente cualquier otro nombre), que se ejecutará continuamente y es por tanto donde se coloca la lógica que controla el movimiento.

Lo primero que se debe hacer es obtener el vector que va desde la cámara hasta la pelota, usando el método "getVectTo" (ver apartado 7.3, página 136), y almacenarlo en la variable "vector_camara_pelota".

A partir del mismo, y teniendo en cuenta sólo las dos primeras componentes (el plano horizontal), se calculan, en función de la acción que corresponda, los vectores necesarios para llevarla a cabo. Por simplicidad se comienza calculando el vector correspondiente al método "force", para obtener el correspondiente al método "torque" a partir de él; observando la Figura 4 puede deducirse que el vector necesario para usar "torque" es siempre perpendicular al necesario para "force" y apuntando a la izquierda, por tanto es sencillo obtener ambos. A continuación se detallan los procedimientos:

- Movimiento hacia adelante y hacia atrás: desde el punto de vista de la cámara, equivale a mover la pelota en la dirección del vector "vector_camara_pelota". En primer lugar se crea un nuevo vector cuyas dos primeras coordenadas sean las mismas que las de "vector_camara_pelota" (cambiadas de signo si el movimiento es hacia atrás) y la tercera 0, llamado "adelante" o "atras" según el caso. Una vez normalizado (para que la fuerza aplicada no dependa de la distancia entre la cámara y la pelota), se obtienen a partir de él los vectores a aplicar a cada método. El correspondiente a "force" será el mismo ("adelante" o "atras"), multiplicado por una constante para controlar la magnitud del efecto, y se llamará "force_adelante" o "force_atras". El vector necesario para el método "torque" es el perpendicular al anterior hacia la izquierda, por tanto se obtiene invirtiendo el orden de sus dos primeras coordenadas y cambiando la primera (la segunda del vector original) de signo. Para aplicarlos se usan

los métodos “applyForce” y “applyTorque” respectivamente, siempre usando coordenadas globales. Ver Fragmento de código 8.

- Movimiento a derecha e izquierda: equivale a mover la pelota perpendicularmente a “vector_camara_pelota”. Se crea un nuevo vector de referencia llamado “perpendicular_izquierda” o “perpendicular_derecha”, cuyas dos primeras coordenadas serán las mismas que las de “vector_camara_pelota” en orden inverso, con una de ellas cambiada de signo en función del sentido deseado (la primera del nuevo vector si es hacia la izquierda, la segunda si es hacia la derecha), y normalizado. El vector necesario para el método “force” será idéntico al de referencia, y el necesario para “torque” será de nuevo perpendicular a él y apuntando hacia la izquierda, obteniéndose como se ha explicado anteriormente. Se aplican de igual manera que en el caso anterior, multiplicándolos por las constantes correspondientes. Ver Fragmento de código 8.

Al generar el movimiento por este método se presenta el problema de que, mientras la tecla está pulsada, en cada cuadro se aplica una cantidad fuerza que se suma a la aplicada en el cuadro anterior, lo que provoca que la velocidad aumente continuamente. De las diferentes soluciones que podrían adoptarse se ha elegido la más simple, que consiste en limitar la velocidad lineal de la pelota. Se hace asignando a su atributo “linVelocityMax” el valor que se desee (en este caso 50) en el nivel principal del módulo: `pelota.linVelocityMax = 50`.

Se ha incluido también la posibilidad de parar en seco la pelota, orientada a hacer más cómodas las comprobaciones del resto de los movimientos. Como sólo se usa durante el desarrollo el realismo no es importante, y por tanto se ha implementado mediante el método “setLinearVelocity”, que modifica la velocidad lineal de los objetos directamente, haciendo que la velocidad sea nula en todas las direcciones. Esta función se ha asignado a la tecla “P”.

En el Fragmento de código 8 se incluye el código correspondiente a los procedimientos explicados en este apartado. Se han omitido los casos de avance hacia atrás y giro a la izquierda por ser análogos al avance hacia adelante y giro a la derecha respectivamente.

```
from bge import logic, events
from mathutils import Vector

teclado = logic.keyboard
raton = logic.mouse
escena_principal = logic.getCurrentScene()
pelota = escena_principal.objects['pelota']
camara_principal = escena_principal.objects['camara_principal']

ACTIVO = logic.KX_INPUT_ACTIVE
RECIEN_ACTIVADO = logic.KX_INPUT_JUST_ACTIVATED
RECIEN_DESACTIVADO = logic.KX_INPUT_JUST_RELEASED

TECLA_GIRO_IZQUIERDA = events.AKEY
TECLA_GIRO_DERECHA = events.DKEY
TECLA_ADELANTE = events.WKEY
TECLA_ATRAS = events.SKEY
TECLA_PARAR = events.PKEY

FACTOR_TORQUE = 15
FACTOR_FORCE = 5

pelota.linVelocityMax = 50

def main():
    vector_camara_pelota =
camara_principal.getVectTo(pelota.position)[1]

    if teclado.events[TECLA_ADELANTE] == ACTIVO:
        adelante =
Vector([vector_camara_pelota[0],vector_camara_pelota[1],0])
        adelante.normalize()
        force_adelante = adelante
        torque_adelante = Vector([-adelante[1], adelante[0],0])
        pelota.applyTorque(FACTOR_TORQUE*torque_adelante,0)
        pelota.applyForce(FACTOR_FORCE*force_adelante,0)

    if teclado.events[TECLA_GIRO_DERECHA] == ACTIVO:
        perpendicular_derecha = Vector([vector_camara_pelota[1],-
vector_camara_pelota[0],0])
        perpendicular_derecha.normalize()
        force_derecha = perpendicular_derecha
        torque_derecha = Vector([-perpendicular_derecha[1],
perpendicular_derecha[0],0])
        pelota.applyTorque(FACTOR_TORQUE*torque_derecha,0)
        pelota.applyForce(FACTOR_FORCE*force_derecha,0)

    if teclado.events[TECLA_PARAR] == ACTIVO:
        pelota.setLinearVelocity([0,0,0],0)
```

Fragmento de código 8. Implementación del movimiento básico de la pelota (en el plano horizontal).

4.5. Movimiento en vertical

El movimiento vertical, que en este apartado se utiliza únicamente para implementar la acción de saltar, es considerablemente más simple que el del plano horizontal, ya que siempre lleva la dirección del eje Z global. La parte compleja es determinar en qué casos debe poderse saltar, que son en principio aquellos en que la pelota esté apoyada sobre una superficie. El código correspondiente a este apartado se ha añadido al del apartado anterior y se ha incluido en un nuevo módulo llamado “control_pelota_2.py”, dentro del fichero de ejemplo

"demo_control_pelota.blend"; basta con cambiar el *script* a ejecutar en el controlador "Python" llamado "control_pelota" del objeto "juego".

Para generar el salto se usa la función "applyImpulse", que aplica un impulso al objeto que la llama en el punto y con la magnitud especificados, de la siguiente manera: `pelota.applyImpulse(punto,direccion_sentido_magnitud)`. Ambos parámetros deben ser vectores de tres dimensiones:

- Punto: indica el punto de aplicación del impulso. Se expresa en coordenadas globales pero tomando como origen el centro del objeto, e influye en su velocidad angular, en la que producirá un cambio siempre que sea distinto de [0,0,0]: por ejemplo, si se da un golpe a un balón de playa en un punto que no esté centrado, comenzará a girar sobre sí mismo. En este caso se aplicará siempre en el origen, para simular el salto con naturalidad.
- Direccion_sentido_magnitud: indica la dirección, el sentido y la magnitud del impulso (equivalente al parámetro pasado al método "applyForce"). Se pasa un vector de la forma [0,0,CONST], donde "CONST" es una constante definida

El efecto de este método, tal y como se usa en el proyecto, es completamente equivalente a usar "applyForce" durante un solo cuadro, por ejemplo respondiendo a una tecla recién activada. Se ha usado simplemente por demostrar el funcionamiento de un método más.

En cuanto a la lógica que determina si debe aplicarse el impulso, consiste simplemente en comprobar si la pelota está apoyada en alguna superficie. Para ello se necesita asociar a la pelota un sensor de tipo "Touch" y acceder a su estado desde el controlador "Python". Dicho controlador puede ser cualquiera, no tiene por qué estar conectado al sensor. En este caso el controlador está asociado al objeto "juego", y el sensor, llamado "contacto_pelota", a "pelota". Lo que sí es imperativo es que el sensor esté conectado a un controlador cualquiera, debido a que Blender no evalúa los sensores si no lo están (en este proyecto se han conectado a controladores "And"). La configuración del sensor queda como se muestra en la Figura 6.

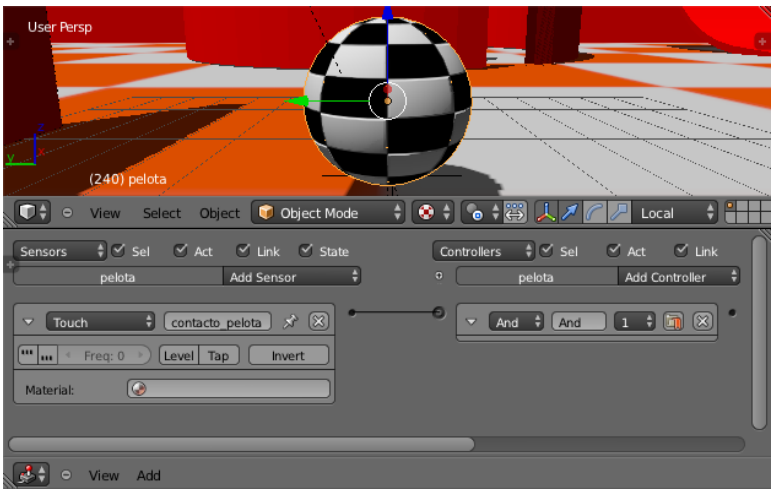


Figura 6. Configuración del sensor "Touch" en el objeto "pelota".

El acceso al sensor se efectúa copiando en una variable la referencia al mismo contenida en la lista de sensores de cada objeto del juego, a la que se accede como a cualquier otro atributo. La selección dentro de dicha lista se hace mediante el nombre de cada sensor. En este caso quedaría: `sensor_contacto_pelota = pelota.sensors['contacto_pelota']`.

Una vez se tiene acceso al sensor, se deben configurar sus parámetros para que responda como sea necesario. Esto se hace modificando sus atributos mediante `nombre_sensor.nombre_atributo`, como se haría con cualquier otro objeto. La lista de atributos y posibles valores se puede consultar en la documentación de Blender. Los parámetros se pueden configurar igualmente en el *logic brick* correspondiente pero, dado que este proyecto pretende estudiar las posibilidades de la programación en Python para controlar Blender, se prefiere hacer mediante Python. De esta manera es posible además cambiar las opciones dinámicamente en función de eventos o variables. Por tanto, al añadir el sensor, el único parámetro que hay que introducir en el *logic brick* es el nombre.

El sensor "Touch" se puede configurar para que sólo se active cuando detecte contacto con un objeto que tenga un determinado material o propiedad, o para que lo haga cuando detecte cualquier contacto. En el caso particular de este apartado, dado que se busca la simplicidad y que el escenario es extremadamente sencillo (las paredes y las rampas tienen el mismo material), se hace que responda siempre independientemente del material o propiedad, dejando todos los valores por defecto; esto implica por una parte que sólo se distingue entre estar en el aire o no, y por otra que en caso de estar la pelota apoyada en una superficie vertical también se permitiría el salto.

La comprobación del estado del sensor puede hacerse de manera análoga a la de las teclas y botones, accediendo a su atributo "status", que tomará valores del 0 al 3 según la pelota esté en el aire, acabe de tocar en el suelo, esté en el suelo o acabe de dejar de tocarlo; también se puede hacer consultando el atributo "positive", que valdrá 0 (False) si el sensor no detecta contacto y 1 (True) si sí lo hace. Como en este caso sólo se requiere distinguir entre dos posibles estados, y de nuevo para demostrar una manera alternativa de conseguir el mismo comportamiento, se hará de la segunda forma.

Por supuesto es necesario también comprobar que la tecla asignada al salto está recién pulsada, quedando la condición final: `if teclado.events[TECLA_SALTAR] == RECIEN_ACTIVADO and sensor_contacto_pelota.positive`. De esta manera la pelota no volverá a saltar si se mantiene la tecla pulsada cuando llegue al suelo. Si la condición fuese por el contrario que la tecla estuviese activa, la pelota saltaría continuamente.

En el Fragmento de código 9 aparecen las modificaciones necesarias para implementar el salto:

```
from bge import logic, events
from mathutils import Vector

#Resto de definiciones de variables y constantes
sensor_contacto_pelota = pelota.sensors['contacto_pelota']
TECLA_SALTAR = events.SPACEKEY
FACTOR_SALTO = 10

def main():

    #Código correspondiente al movimiento en el plano horizontal

    if teclado.events[TECLA_SALTAR] == RECIEN_ACTIVADO and
sensor_contacto_pelota.positive:
        pelota.applyImpulse([0,0,0],[0,0,FACTOR_SALTO])
```

Fragmento de código 9. Código a añadir al módulo de control del movimiento de la pelota para implementar el salto

4.6. Añadidos

Para complementar lo anterior se han añadido algunas modificaciones a los controles básicos, que en un juego real podrían responder por ejemplo a trucos o logros:

4.6.1. Aumentar la velocidad al pulsar una tecla

Consiste simplemente en comprobar si la tecla elegida está pulsada, y en ese caso multiplicar los vectores que se aplican al método "applyTorque" por un número más grande que lo habitual.

Una posible forma de hacerlo sería colocar, una vez dentro de cada uno de los bloques correspondientes al movimiento básico horizontal, una estructura if/else: si la tecla asignada al aumento de la velocidad está pulsada aplicar una cantidad de "torque", si no aplicar otra. Sin embargo se ha optado por hacer dicha comprobación en el inicio de la función "main", y en función del estado de la tecla asignar a la constante que gobierna la magnitud del "torque" aplicado un valor u otro.

Concretamente se ha añadido una nueva constante, definida en el nivel superior (llamada "FACTOR_TURBO"), que contiene el factor por el que se multiplica el "torque" aplicado cuando se pulsa la tecla. En la función "main" se han cambiado las constantes que se aplican en el método "applyTorque" de "FACTOR_TORQUE" a "FACTOR_TORQUE_FINAL", pudiendo ser su valor equivalente a "FACTOR_TORQUE" o a "FACTOR_TORQUE * FACTOR_TURBO" en función del estado de la tecla correspondiente.

Nótese que de la misma manera que se ha modificado el valor aplicado a "torque" se podría haber cambiado el de "force" o una combinación de ellos, y en vez de responder a una tecla podría hacerlo a cualquier otro evento, siendo las posibilidades muy amplias.

En el Fragmento de código 10 se reflejan los cambios que se han hecho respecto a la situación anterior. El código completo se encuentra en el módulo "control_pelota_3.py",

presente, en el fichero de ejemplo “demo_control_pelota.blend”; como en apartados anteriores, se puede configurar el controlador “Python” correspondiente al control de la pelota para que lo lance (“control_pelota_3.main”) y así comprobar el funcionamiento y hacer los cambios que se deseen.

```
from bge import logic, events
from mathutils import Vector

# Resto de definiciones de variables y constantes

TECLA_TURBO = events.RIGHTSHIFTKEY

FACTOR_TORQUE = 15
FACTOR_TURBO = 3

def main():

    if teclado.events[TECLA_TURBO] == ACTIVO:
        FACTOR_TORQUE_FINAL = FACTOR_TURBO * FACTOR_TORQUE
    else:
        FACTOR_TORQUE_FINAL = FACTOR_TORQUE

    # Resto del código correspondiente al movimiento en el plano
horizontal

     if teclado.events[TECLA_ADELANTE] == ACTIVO:
        adelante =
Vector([vector_camara_pelota[0],vector_camara_pelota[1],0])
        adelante.normalize()
        force_adelante = adelante
        torque_adelante = Vector([-adelante[1], adelante[0],0])
        pelota.applyTorque(FACTOR_TORQUE_FINAL*torque_adelante,0)
        pelota.applyForce(FACTOR_FORCE*force_adelante,0)
```

Fragmento de código 10. Código a añadir para implementar una función de “turbo”.

4.6.2. Hacer posible el control total en las tres dimensiones

Consiste básicamente en permitir controlar el objeto protagonista en la dirección vertical, usando para ello dos nuevas teclas. En el caso de este proyecto se ha implementado como un modo de control que no está disponible por defecto, sino que es necesario activar y desactivar pulsando una secuencia de teclas (al estilo de los trucos de algunos juegos comerciales). Para hacer más fácil el control, mientras el modo está activo se elimina la gravedad, de manera que sobre la pelota solo actúan las fuerzas que el usuario desea; asimismo se incrementa la fuerza aplicada en todos los ejes para poder realizar los movimientos con mayor rapidez.

En este apartado sólo se cubre la parte que atañe directamente al control de la pelota (el código que captura secuencias de teclas y el que modificar la gravedad se encuentran en el módulo “otros.py”). Para ello sólo es necesario saber que la detección la palabra clave (en este caso “volar”) origina un cambio en la propiedad “volar” del objeto “pelota”, pasando de 0 a 1 o viceversa.

La generación del movimiento hacia arriba y hacia abajo es relativamente simple ya que, al igual que en el apartado 4.5 (página 38), siempre se desarrolla en el eje Z global, por lo que

no es necesario hacer cálculos. El método usado es “applyForce”, ya que cuando la pelota se mueve por el aire no existe rozamiento alguno que pueda convertir el giro que se consigue con “applyTorque” en movimiento absoluto.

En cuanto al código necesario (incluido en el módulo “control_pelota_4.py”) puede dividirse en dos partes con funciones diferentes (aunque relacionadas entre sí):

- Incrementar el valor de la constante que controla el método “applyForce”: se sigue el mismo razonamiento que en el apartado anterior, donde se modificaba la constante aplicada a “applyTorque”. En las llamadas al método se cambia la constante “FACTOR_FORCE” por “FACTOR_FORCE_FINAL”; el valor de esta última se determina al principio de la función “main”, pudiendo ser equivalente a “FACTOR_FORCE_VOLAR” (definida en el nivel superior) o a “FACTOR_FORCE” según el contenido de la propiedad “volar” del objeto “pelota” (ver Fragmento de código 11).
- Generar el movimiento vertical: dado que sólo se permite cuando se ha introducido la palabra clave, se aprovecha para invocarlo en el mismo bloque de código donde, en el apartado anterior, se asignaba un valor u otro a la constante “FACTOR_FORCE_FINAL”. Es decir, consiste en comprobar primeramente si la propiedad “volar” vale 1, y en ese caso si alguna de las teclas correspondientes está pulsada; si es así, se aplica “applyForce” en el eje Z usando la constante anterior (ver Fragmento de código 11)

En el Fragmento de código 11 se reflejan los cambios necesarios respecto al código anterior para hacer posible el movimiento en vertical y aumentar la fuerza aplicada en el horizontal. En cuanto a este último, sólo se ha incluido el bloque de código correspondiente al movimiento hacia adelante, por ser el resto equivalentes.

En el fichero de ejemplo, estos cambios se han incluido en el módulo “control_pelota_4.py”; para usarlo es necesario configurar el controlador “Python” correspondiente para que lance su función “main” (“control_camara_4.main”).

```
from bge import logic, events
from mathutils import Vector

# Resto de definiciones de variables y constantes
TECLA_ARRIBA = events.UPARROWKEY
TECLA_ABAJO = events.DOWNARROWKEY

FACTOR_FORCE = 5
FACTOR_FORCE_VOLAR = 30

def main():

    if 'volar' in pelota and pelota['volar'] == 1:
        FACTOR_FORCE_FINAL = FACTOR_FORCE_VOLAR

        if teclado.events[TECLA_ARRIBA]==ACTIVO:
            pelota.applyForce([0,0,FACTOR_FORCE_FINAL],0)

        if teclado.events[TECLA_ABAJO]==ACTIVO:
            pelota.applyForce([0,0,-FACTOR_FORCE_FINAL],0)
    else:
        FACTOR_FORCE_FINAL = FACTOR_FORCE

    # Resto del código correspondiente al movimiento en el plano
horizontal

    if teclado.events[TECLA_ADELANTE] == ACTIVO:
        adelante =
Vector([vector_camara_pelota[0],vector_camara_pelota[1],0])
        adelante.normalize()
        force_adelante = adelante
        torque_adelante = Vector([-adelante[1], adelante[0],0])
        pelota.applyTorque(FACTOR_TORQUE_FINAL*torque_adelante,0)
        pelota.applyForce(FACTOR_FORCE_FINAL*force_adelante,0)
```

Fragmento de código 11. Código a añadir para implementar el movimiento en el eje vertical y aumentar la magnitud de la fuerza aplicada.

5. CONTROL DE LA CÁMARA

5.1. Introducción

El objetivo buscado en este bloque es conseguir un sistema de seguimiento del objeto protagonista que se asemeje al de algunos juegos comerciales, donde el usuario no necesita interactuar con él salvo que desee un punto de vista específico. Es decir, se pretende que haya un sistema de control de la cámara automático, que coloque la cámara de forma que en todo momento se visualice la acción correctamente, y uno manual, que se accione cuando el usuario quiera mover la cámara y actúe en consecuencia (en este caso moviendo el ratón).

En los siguientes apartados se pretende explicar escalonadamente y de forma organizada cómo se han implementado estos sistemas. En primer lugar se tratará el control automático, que tiene efecto en el plano horizontal; seguidamente se introducirá el control con el ratón en dicho plano, y por último se ampliará al plano vertical.

El fichero donde se encontrarán los módulos a los que se hace referencia es el llamado "demo_control_camara.blend". Se debe configurar el uso del módulo adecuado para cada apartado en el controlador "Python" correspondiente, asociado al objeto "juego", de tipo "Empty", situado en el centro de la escena principal.

Al abrir dicho fichero y ejecutar el juego se observará que se ha añadido otra ventana a además del mini-mapa de orientación. En ella se muestra la imagen captada por una cámara auxiliar, en la que inicialmente aparecen tanto la pelota como unos prismas alargados que representan a la cámara "camara_principal" (el punto donde se juntan) y a sus ejes (la correspondencia entre ejes y prismas se ha establecido asignando a cada eje un material del color estándar de Blender, y a cada sentido, excepto al negativo del eje Z, una luminosidad diferente). Su finalidad es permitir visualizar los movimientos de la cámara principal en tiempo real, y comprobar el funcionamiento de los diferentes apartados.

El control de esta cámara auxiliar se lleva a cabo desde el módulo "camara_auxiliar.py". El proceso de añadir una ventana con su imagen es idéntico al del caso del mini-mapa, cubierto en el apartado 1 (página 107), y el movimiento de la cámara en sí se ha implementado de la manera más simple posible. Se ha hecho que siempre enfoque bien al punto medio entre la cámara y la pelota, o bien a una de ellas; pulsando la tecla C se cambia entre ellos. Mediante las flechas del teclado se mueve hacia arriba, abajo, izquierda o derecha, y mediante las teclas +/- se acerca aleja o acerca al punto de enfoque.

5.2. Generación del movimiento

Al contrario que en el caso del objeto protagonista, donde se buscan movimientos naturales y realistas, y por ello el movimiento se genera mediante la aplicación de fuerzas (dejando que el motor físico de Blender calcule la velocidad correspondiente en cada caso) en el manejo de la cámara no interviene el concepto de realismo: por lo general, y especialmente en situaciones de tercera persona, en juegos y películas la cámara se sitúa y mueve de una

manera totalmente diferente a como lo harían los ojos de una persona que estuviese viendo la escena en directo. Lo que se persigue a la hora de mover la cámara es que proporcione una visión adecuada de la acción y que sus movimientos no resulten molestos.

La técnica más extendida para controlar la cámara consiste en emparentarla con un objeto invisible, normalmente de tipo "Empty", que a su vez se emparenta al centro del objeto protagonista, de manera que sigue sus movimientos (generalmente sólo sigue el desplazamiento, mientras que la rotación se controla independientemente, lo que se consigue estableciendo un parentesco de tipo "Vertex"). De esta forma, el movimiento de la cámara se genera aplicando rotaciones al objeto con el que está emparentada: por ejemplo, rotarlo sobre el eje vertical ocasionaría que la cámara se moviera a izquierda y derecha.

En este proyecto, para seguir con la filosofía de usar Python siempre que sea posible, estas relaciones de parentesco no se han establecido en la escena, sino que se han imitado mediante la aplicación de velocidad y rotaciones sobre sí misma a la cámara. Además, esto ha llevado al desarrollo de un sistema que puede adaptarse fácilmente al control de cualquier sólido rígido (más adelante se trata el tipo físico del objeto cámara), y cuya flexibilidad es mucho mayor, y ha permitido explorar aspectos que de otra forma no habrían aparecido.

En todo caso el uso de fuerzas, como en el caso del objeto protagonista, queda descartado: tanto aplicar velocidades a la cámara como usar parentescos y rotaciones permiten controlar con mucha más precisión los movimientos que se producen. Por ejemplo, en los casos en que es necesario detener la cámara en seco, sería mucho más complicado calcular las fuerzas necesarias para que la velocidad resultante al aplicarlas sea cero, que directamente aplicar una velocidad o rotación nula.

En cuanto a la imitación del parentesco entre la cámara y un objeto que rota, al control del desplazamiento mediante velocidades se le añade la complejidad de que es necesario modificar también su orientación. Tomando como referencia la Figura 7 puede observarse cómo para que la cámara se mueva hacia la derecha según su punto de vista es necesario, en primer lugar, que se desplace hacia la derecha, y, en segundo lugar, que rote hacia la izquierda sobre el eje vertical para seguir apuntando hacia la pelota. Esto ocurre igualmente al realizar movimientos en el resto de las dimensiones.

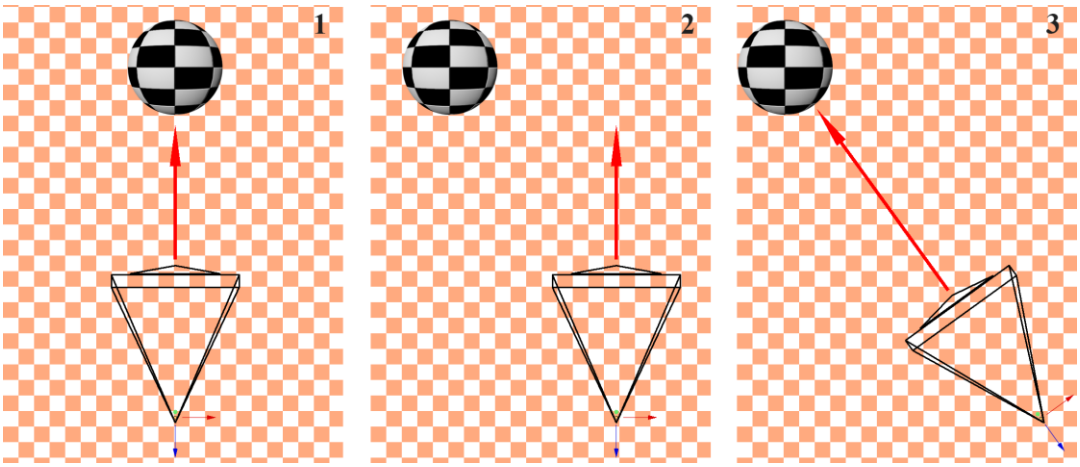


Figura 7. Fundamentos del movimiento de la cámara. 1: situación inicial, 2: desplazamiento, 3: rotación.

5.2.1. Tipo físico del objeto "camara_principal"

Para poder controlar el movimiento aplicando velocidades es imprescindible que el tipo físico del objeto sobre el que se aplican sea "Rigid Body" o "Dynamic" (en este caso se ha definido como sólido rígido). Esto implica que el motor físico de Blender calcula la posición de dicho objeto en función de los parámetros definidos en la pestaña "Physics", de la gravedad y de las fuerzas o velocidades aplicadas.

La consecuencia más inmediata de esto es que es necesario controlar continuamente su posición vertical, para compensar el efecto de la gravedad. De lo contrario la cámara caería al suelo nada más lanzar el juego

En cuanto a los parámetros de la pestaña "Physics" solamente es necesario activar la casilla "Ghost", para evitar que la cámara interactúe con el resto de objetos. Podría parecer conveniente que chocase contra las paredes, de manera que parase al llegar a una de ellas mientras avanza hacia atrás, y así evitar que la pared se interpusiera entre la cámara y la pelota. Sin embargo esto constituye un problema en la situación contraria: si la cámara choca contra una pared de frente, por ejemplo al doblar la pelota una esquina rápidamente, se queda sin poder avanzar y sin visión de la escena. Los choques contra las paredes se tratarán más adelante mediante Python, con la ayuda de un sensor "Near".

La casilla "Actor" se deja sin marcar en este caso, ya que en el juego no se usa ningún sensor "Near" ni "Radar" que deba detectar la presencia de la cámara.

5.2.2. Desplazamiento

Dado que se ha decidido aplicar velocidades para desplazar la cámara, el método apropiado es "setLinearVelocity". Este método recibe como parámetros un vector tridimensional con la velocidad a aplicar y una variable booleana (True / False), que indica si se deben tener en cuenta las coordenadas locales de la cámara o las globales al interpretar el vector.

Al contrario que en la pelota, los ejes locales de la cámara permanecen siempre en una posición conocida. Esto, junto con el hecho de que la orientación de los ejes se corresponde con direcciones útiles para la generación de movimiento (ver Figura 8), hace que sea más práctico en un principio usar las coordenadas locales. La correspondencia entre las direcciones de interés y los ejes de la cámara es la siguiente:

- Velocidad lateral: se denomina así a la magnitud que se coloca en la primera componente del vector aplicado en el método "setLinearVelocity", es decir, la que corresponde al eje X. En la Figura 8 se corresponde con las flechas rojas. Es responsable, junto a la rotación, del giro alrededor de la pelota. Ejemplo: `setLinearVelocity([1,0,0],1)` ocasiona que la cámara se mueva hacia la derecha.
- Velocidad vertical: se aplica al eje Y de la cámara (segunda coordenada del vector velocidad), y ocasiona desplazamientos hacia arriba y abajo. En la Figura 8 está representada por flechas verdes. Ejemplo: `setLinearVelocity([0,1,0],1)` ocasiona que la cámara se mueva hacia arriba.
- Velocidad lineal: la que ocasiona que la cámara se acerque o aleje de la pelota, representada en la Figura 8 por flechas azules. Corresponde al eje Z de la cámara (nótese que la parte positiva de este eje apunta hacia la parte trasera de la cámara). Ejemplo: `setLinearVelocity([0,0,1],1)` ocasiona que la cámara se aleje de la pelota.

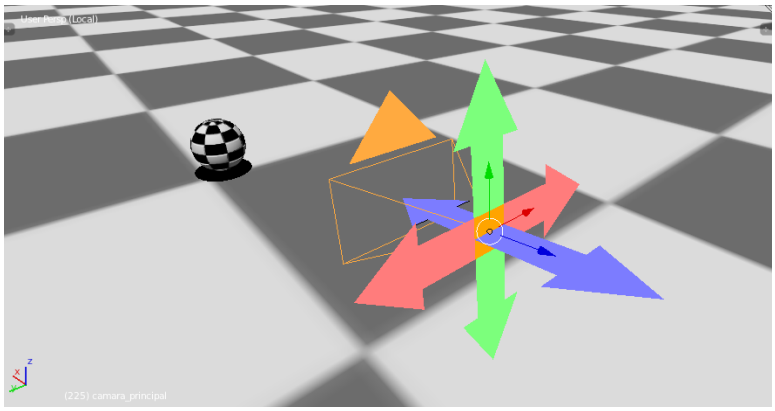


Figura 8. Correspondencia entre los ejes de la cámara y los movimientos que se le aplican. Rojo: eje X / velocidad lateral, verde: eje Y / velocidad vertical, azul: eje Z / velocidad lineal.

El desplazamiento necesario en cada dirección se calcula en cada cuadro y por separado, en función de las variables correspondientes. Por ejemplo, si en un instante es necesario dejar de mover la cámara lateralmente, acercarla a la pelota y hacer que se eleve ligeramente, se podrían tener las variables: velocidad_lateral = 0, velocidad_lineal = -2, velocidad_vertical = 0.5. Como la dirección en la que hay que mover la cámara para que se desplace lateralmente es su eje X, y lo mismo ocurre con el resto, siendo la correspondencia la indicada anteriormente, en ese cuadro el comando necesario sería: `camara_principal.setLinearVelocity([0, 0.5, -2],1).`

Es importante tener en cuenta que la asociación entre los ejes de la cámara y las direcciones del espacio (lateral, vertical, arriba, abajo) es simplemente una ayuda a la programación. Movimientos en el eje Y de la cámara, que se relacionará siempre con la magnitud "velocidad_vertical", sí la desplazarán arriba y abajo si su posición no es elevada respecto a la pelota; sin embargo en caso contrario, como la cámara rota para apuntar hacia la pelota, sus ejes locales rotan con ella, y a partir de cierto punto (cuando su eje Z forme un ángulo mayor de 45º con el plano horizontal) aplicar velocidad en el eje Y implicará desplazarse hacia delante y atrás en mayor medida que hacia arriba y abajo.

Relacionado con este detalle aparece el hecho de que, aún con la cámara situada en la posición inicial, siempre que se actúe sobre su eje Y local se ocasionará un pequeño desplazamiento hacia adelante o atrás, ya que éste no coincide con el eje Z global. Lo mismo ocurrirá con el eje Z local: al desplazar la cámara hacia adelante para seguir a la pelota, dado que este eje no es paralelo al plano horizontal, se desplazará también hacia abajo. Estas influencias entre movimientos no serán importantes en las primeras etapas de desarrollo, sin embargo finalmente harán necesario cambiar la forma en la que se aplica la velocidad para pasar a hacerlo con referencia a coordenadas globales y poder controlar cada movimiento por separado.

5.2.3. Rotación

Como se ha comentado anteriormente, siempre que la posición relativa entre la cámara y la pelota cambie será necesario rotar la cámara para que siga apuntando hacia la ella.

En lugar de modificar manualmente la orientación, en este caso se usa el método "alignAxisToVect", presente en todos los objetos del juego, que alinea uno de los ejes del objeto desde el que se llama con un vector tridimensional dado. Este método toma los siguientes parámetros: el vector al cual se quiere alinear el eje, un número entero que representa qué eje alinear (0: eje X, 1: eje Y, 2: eje Z) y un número entre 0 y 1 que determina la fracción del cambio total que se hace. Este último parámetro tiene el mismo efecto que el suavizado que se ha implementado manualmente en otras partes del proyecto (explicado en el apartado 7.6): por ejemplo, si el eje X de un objeto necesita hacer un giro de 90º para estar alineado con el vector dado, y el parámetro vale 0.5, se hará un giro de 45º. En el siguiente cuadro el giro necesario será de 45º por lo que se hará de 22.5º, y así sucesivamente.

En cuanto al eje que debe ser alineado, hay que tener en cuenta lo siguiente:

- Si se alinea el eje Z (su parte negativa) con el vector que va desde la cámara hasta la pelota (vector_camara_pelota), opción que a priori podría parecer suficiente, la cámara es libre de rotar alrededor de dicho eje, como se puede observar en la Figura 8. En la práctica esto lleva a que la cámara se ladea tan pronto como la pelota cambia de dirección, llegando a girar completamente y verse la imagen "boca abajo".
- Si se alinea el eje X con un vector perpendicular al que une la cámara y la pelota (vector_camara_pelota), y paralelo al plano horizontal (plano XY), se soluciona el problema anterior. Sin embargo, de esta forma se permite que gire sin control

alrededor del eje X, lo que conlleva un comportamiento si cabe más negativo, ya que la pelota puede incluso desaparecer del campo de visión, y por tanto tampoco es válido.

A la vista de lo anterior, la solución es hacer ambas operaciones en cada cuadro. Mediante la primera se coloca la pelota en el centro de la pantalla, y mediante la segunda se evita que la cámara se ladee. El vector requerido en el segundo caso es uno tridimensional cuyas dos primeras componentes son las de "vector_camara_pelota", cambiadas de orden y la segunda de signo, y la tercera (Z) nula. En el Fragmento de código 12 aparecen las líneas de código necesarias en cada cuadro para orientar la cámara correctamente (el vector perpendicular a "vector_camara_pelota" hacia la derecha se ha definido aparte para hacer el proceso más claro):

```
# Asignaciones del nivel principal...

def main():
    #Resto de lógica...

    vector_camara_pelota =
camara_principal.getVectTo(pelota.position)[1]

    camara_principal.alignAxisToVect(-vector_camara_pelota,2,1)
    perpendicular_derecha = [vector_camara_pelota[1],-
vector_camara_pelota[0],0]
    camara_principal.alignAxisToVect(perpendicular_derecha,0,1)
```

Fragmento de código 12. Orientación de la cámara en cada cuadro para que apunte hacia la pelota.

Es importante entender el resultado de la aplicación conjunta de desplazamiento y rotación: dado que el desplazamiento se calcula sobre los ejes locales, y que la orientación de éstos cambia cada vez que se produce una rotación (ver Figura 7), el efecto final es que la cámara gira alrededor de la pelota a modo de movimiento orbital, tendiendo a situarse siempre en una esfera de radio la distancia inicial entre ella y la pelota.

5.3. Seguimiento básico del objeto protagonista

En este apartado se explica cómo se ha implementado el seguimiento más básico de la pelota. Éste se basa en tres premisas fundamentales:

- Control de la posición vertical: para contrarrestar el efecto de la gravedad. Podrá ser constante o definirse en función de la posición de la pelota.
- Mantener la pelota en el centro de la pantalla: rotando la cámara, usando el método explicado en el apartado 5.2.3.
- Mantener la distancia entre la cámara y la pelota: aplicando una cantidad de velocidad en el eje Z proporcional a la misma en cada instante.

Para seguir lo explicado en los siguientes sub-apartados es conveniente configurar el fichero de referencia ("demo_control_camara.blend") para que el controlador "Python"

correspondiente lance el módulo "control_camara_1.py". Los bloques de código presentes en dicho módulo que se corresponden con estos sub-apartados están en un principio encerrados entre comillas triples (desactivados), de forma que según se vaya avanzando en la explicación se vayan activando para observar su efecto aislado.

5.3.1. Control de la posición vertical

Como se ha comentado anteriormente, el hecho de que la cámara sea un sólido rígido conlleva que se ve afectada por la gravedad, lo que hace necesario intervenir para mantener estable su posición vertical.

La solución más lógica y simple a priori sería modificar su posición en cada cuadro, mediante una línea del tipo: `camara_principal.worldPosition[2] = altura_correspondiente`. Sin embargo, esto resulta en que la cámara se desplaza continuamente hacia abajo, muy lentamente. Esto probablemente se deba a un conflicto entre la actuación del motor físico, que calcula la velocidad debida a la gravedad, y el establecimiento de la posición manualmente.

El mismo comportamiento se repite si en lugar de definir la posición se aplica una velocidad vertical nula, mediante: `camara_principal.setLinearVelocity([n,0,n], 1)` (Recuérdese la disposición de los ejes locales de las cámaras, Figura 8). Por tanto tampoco este método es viable.

La manera más simple de mantener la cámara en la misma posición vertical es combinar los dos métodos anteriores. Si se establece una posición determinada, y acto seguido se hace nula la velocidad en el eje vertical, la cámara no se mueve.

Sin embargo, ya que en los apartados donde intervenga el control manual (con el ratón) en vertical será necesario efectuarlo aplicando velocidades, es decir, se aplicará una velocidad vertical distinta de 0, en este caso se hace de la misma manera: se usa solamente el método "setLinearVelocity", aplicando la velocidad vertical necesaria para contrarrestar el efecto de la gravedad.

El valor que debe tomar dicha velocidad vertical se calcula siguiendo el mismo principio que en otros apartados: se busca una relación óptima entre el mismo y la/las magnitudes de las que se desea que dependa, mediante operaciones matemáticas simples. En este caso, la velocidad vertical a aplicar se relaciona con la diferencia en cada momento entre la posición vertical de la pelota y la que debería tener.

Hay que tener en cuenta la distribución de los ejes del objeto sobre el que se actúa, para hacer posibles cambios de signo o dirección. En este caso se puede razonar como sigue: la posición vertical deseada para la cámara, que en principio será estática, siempre es mayor que la que tiende a adoptar en cada cuadro, ya que la gravedad la empuja hacia abajo. Por tanto la diferencia entre ellas (deseada – actual) es positiva. El efecto de la velocidad aplicada debe ser contrario al de la gravedad; como ésta empuja a la cámara hacia el sentido negativo del eje Y, la velocidad debe aplicarse en el positivo.

Por otra parte, si fuera necesario hacer descender expresamente a la cámara la posición deseada sería menor que la actual, con lo cual la diferencia menor que cero, y la velocidad a aplicar en el eje Y debería ser también negativa. Existe por tanto una correspondencia entre el signo de dicha diferencia y el de la velocidad necesaria, con lo cual no habrá que tratar ambos casos por separado.

Sólo resta decidir cómo hay que transformar la diferencia entre posiciones para que la velocidad resultante sea adecuada. Esto se hace simplemente probando diferentes operaciones. La más sencilla, suficiente en este caso, es la multiplicación por un determinado factor. En otras ocasiones, que se estudiarán posteriormente, hará falta adaptar los rangos de variación de las magnitudes implicadas, o modificar la función de transferencia. Destacar que no hay una sola manera de establecer la relación. De hecho el cómo se haga determina el comportamiento resultante.

En este caso, como se ha indicado, la velocidad vertical aplicada consiste en la diferencia entre la posición vertical deseada de la cámara y la actual multiplicada por un determinado factor. Este factor, dentro de su rango válido, determina la rapidez con la que se consigue la posición deseada. Debe ser lo suficientemente grande como para que la velocidad aplicada contrarreste a la gravedad, y lo suficientemente pequeño como para que no se produzcan inestabilidades.

En el Fragmento de código 13 aparece el ejemplo más sencillo, que simplemente mantiene la posición vertical de la cámara estática en todo momento. Nótese como en este caso la posición deseada de la cámara es la que tiene inicialmente. Se recomienda ejecutar el archivo “demo_control_pelota.blend”, y probar con diferentes valores del factor anterior, viendo su efecto en cada ocasión sobre la velocidad aplicada y la posición de la cámara. Para ello, el controlador “Python” correspondiente debe estar configurado para lanzar “control_camara_1.main”, y en este módulo deben estar anulados todos los bloques de código de la función “main” excepto el primero.

```
from bge import logic

escena_principal = logic.getCurrentScene()
camara_principal = escena_principal.objects['camara_principal']

altura_camara_inicial = camara_principal.position[2]

def main():
    altura_camara_actual = camara_principal.position[2]
    altura_camara_deseada = altura_camara_inicial

    velocidad_vertical = 2 * (altura_camara_deseada -
altura_camara_actual)
    camara_principal.setLinearVelocity([0,velocidad_vertical,0],1)
```

Fragmento de código 13. Aplicar velocidad vertical a la cámara para contrarrestar el efecto de la gravedad.

Un planteamiento algo más complejo, y que será el definitivo, es establecer la posición de la cámara en función de la del objeto protagonista, de manera que reaccione a sus posibles

cambios de altura. En este caso la altura deseada es la de la pelota más la diferencia de altura entre ella y la cámara que haya inicialmente, cálculo que habrá que hacer al principio de la ejecución. El resto del razonamiento es análogo al del caso anterior, incluyendo el criterio de símbolos.

En el Fragmento de código 14 aparece el código que habría que cambiar o añadir (se obvia la lógica que no cambia respecto a fragmentos de código anteriores) para conseguir este comportamiento. Si se ejecuta el fichero de ejemplo, activando el siguiente bloque de código y desactivando el anterior, se observará cómo efectivamente la cámara se mueve hacia arriba y abajo cuando la pelota salta. La rapidez con la que se adapta a sus movimientos está determinada por el factor que multiplica a la diferencia de alturas, 5 en este caso.

```
# Resto de asignaciones...

pelota = escena_principal.objects['pelota']
altura_camara_pelota_inicial = camara_principal.position[2] -
pelota.position[2]

def main():
    altura_camara_actual = camara_principal.position[2]
    altura_camara_deseada = pelota.position[2] +
altura_camara_pelota_inicial

    velocidad_vertical = 5 * (altura_camara_deseada -
altura_camara_actual)
    camara_principal.setLinearVelocity([0,velocidad_vertical,0],1)
```

Fragmento de código 14. Determinar la posición vertical de la cámara en función de la del objeto protagonista.

Nótese cómo, en los dos casos anteriores, se han basado los cálculos que se hacen en cada cuadro en la situación de la escena al empezar la ejecución. Esto conlleva que se pueden llevar a cabo modificaciones manuales en la escena que se reflejarán en el juego sin la necesidad de cambiar el código. Por ejemplo, si antes de lanzar el juego se coloca la cámara en una posición más elevada, esa será la posición objetivo que se tienda a adoptar durante el mismo. Esta manera de proceder se intentará seguir siempre que sea posible.

En el fichero de texto aparece un tercer bloque de código, llamado “Calcular velocidad vertical (lógica definitiva)”, equivalente al anterior excepto por la última línea, que se ha eliminado. Este bloque se deberá activar (y desactivar el anterior) más adelante, cuando se implemente la velocidad lineal, para evitar que se aplique un valor nulo a la misma. Sería suficiente con eliminar la línea que aplica la velocidad, pero se ha considerado más claro repetir el bloque entero.

5.3.2. Rotación sobre sí misma para mantener al objeto protagonista en el centro de la pantalla

Es otro de los pilares en los que se basa el movimiento de la cámara, responsable junto a la aplicación de velocidad lineal de que ésta siga los pasos del objeto protagonista. Consiste

en seguir las explicaciones del apartado 5.2.3: usando el método "alignAxisToVect", alinear su eje Z negativo con el vector que une cámara y pelota (denominado "direccion_deseada"), para que la pelota se coloque en el centro de la pantalla, y hacer lo mismo con el eje X, esta vez alineándolo con un vector perpendicular al anterior apuntando hacia la derecha y paralelo al plano horizontal, para que la cámara no se incline hacia los lados.

En cuanto al tercer parámetro que se pasa a este método, cuyo efecto es el de no llevar a cabo la alineación inmediatamente, sino sólo una parte de ella, y así conseguir un suavizado, hay que tener en cuenta que el método se aplica dos veces seguidas, y sus efectos no son independientes uno del otro. El comportamiento ideal sería que solamente se suavizase la parte que mantiene la pelota centrada, mientras que la que hace que la cámara no se ladee debería actuar inmediatamente. Sin embargo esto no es posible: la operación necesaria para mantener la horizontalidad, alinear el eje X al vector descrito anteriormente, ocasiona que todo el objeto se mueva, y con ello el eje Z, que instantáneamente pasa a apuntar hacia algún punto de la vertical de la pelota, con lo que se anula la parte horizontal del suavizado perseguido en la primera aplicación del método. Sin embargo, como el eje alineado en esta segunda aplicación es el X, no afecta a la parte vertical del suavizado que se ha introducido en la primera. Para conseguir un suavizado completo habría que usar un factor menor que 1 en ambas operaciones, lo que provocaría que la cámara se ladease ligeramente en cada giro de la pelota (lo que por otra parte podría ser admisible).

Un efecto parecido al que se conseguiría idealmente con el método anterior se puede implementar de la siguiente manera: en cada momento, apuntar la cámara no hacia la posición actual de la pelota sino a la que ocupaba unos cuadros antes. Para ello hay que hacer uso de la propiedad "historial_posicion_pelota" del objeto "pelota", explicada en la primera parte del apartado 7.7. En esta propiedad está almacenada una cola que contiene sus últimas posiciones. Simplemente es necesario, al llamar al método "getVectTo" de la cámara, pasarle como parámetro uno de los elementos de la lista en lugar de "pelota.position". El elemento elegido determinará el retraso que se obtiene. Dado que la lista de posiciones es de tipo "deque" (ver apartado 7.4, página 137), la posición más reciente (la actual, no produce retraso) será la última, y la más antigua la primera (retraso máximo). Si se desea un retraso mayor que el conseguido usando el primer elemento (índice 0), basta con usar una cola de más capacidad, haciendo el cambio correspondiente en la definición de la misma en el módulo "logica_comun.py".

Este método tiene el inconveniente de que los cambios bruscos en la velocidad de la pelota (por ejemplo choques contra paredes o saltos) provocan cambios igualmente bruscos pero retrasados en la velocidad de rotación de la cámara. Los cambios en la cámara no se perciben si ocurren al mismo tiempo que los de la pelota, pero si están retrasados sí se apreciará un comportamiento extraño.

El procedimiento que se siga o la combinación de ellos dependerá de las necesidades y preferencias en cada caso. En este proyecto se han descartado tanto la introducción de retraso como el uso de suavizado: a priori podría aprovecharse el suavizado vertical que se consigue

pasando un número menor que 1 como último parámetro al alinear el eje Z, pero éste constituirá un problema cuando se introduzca el movimiento en vertical de la cámara.

En el Fragmento de código 15 aparece el código a añadir para efectuar la rotación incluyendo retraso (aunque en este caso es nulo -se elige el último elemento de "historial_posicion_pelota"-, se ha colocado la estructura correspondiente para demostrar cómo se implementaría si fuera necesario). La asignación `historial_posicion_pelota = pelota['historial_posicion_pelota']` se hace solamente por comodidad.

Para comprobar el funcionamiento en el fichero de ejemplo se debe activar el último bloque de código, llamado "Apuntar la cámara hacia la pelota".

```
# Resto de asignaciones...

historial_posicion_pelota = pelota['historial_posicion_pelota']

def main():
    # Resto de logica...

    direccion_deseada =
camara_principal.getVectTo(historial_posicion_pelota[-1])[1]

    camara_principal.alignAxisToVect(-direccion_deseada,2,1)
    perpendicular_derecha = Vector([direccion_deseada[1],-
direccion_deseada[0],0])
    camara_principal.alignAxisToVect(perpendicular_derecha,0,1)
```

Fragmento de código 15. Rotación de la cámara para que apunte siempre hacia el objeto protagonista.

5.3.3. Avance y retroceso para mantener la distancia entre la cámara y la pelota

Consiste en aplicar en cada momento la velocidad necesaria para que la distancia entre la cámara y la pelota se mantenga constante. Esta velocidad se aplica directamente al eje Z de la cámara (ver apartado 5.2.2, página 47), y se denomina "velocidad lineal".

El principio que se sigue para calcular la velocidad necesaria es similar al explicado en el caso de la velocidad vertical (apartado 5.3.1): se aplica una velocidad proporcional a la diferencia entre la distancia objetivo y la actual, de manera que su efecto reduzca dicha diferencia. La distancia objetivo es de nuevo la que hay entre ambos objetos al comenzar el juego. Para calcular las distancias se usa el método "getVectTo" (ver apartado 7.3, página 136).

Dado que es el sentido negativo del eje Z de la cámara el que apunta hacia la pelota, cualquier velocidad positiva hará que la distancia entre ambas aumente, y viceversa.

Cuando esta distancia (actual) es menor que la inicial (deseada) la diferencia entre ambas, si se calcula como deseada - actual, es positiva. Al aplicar directamente como velocidad dicha diferencia o un valor basado en ella (multiplicada por un factor mayor que cero), se consigue el resultado deseado. Si la situación es la inversa, la relación sigue siendo válida.

En el Fragmento de código 16 se muestra el código a añadir para implementar esta funcionalidad. En este caso la diferencia se ha multiplicado por el factor 2, pero de nuevo es conveniente probar diferentes valores para observar su funcionamiento. En el fichero de ejemplo se debe activar el bloque de código correspondiente a este apartado (llamado "Calcular velocidad lineal"), cambiar el bloque que calcula la velocidad vertical (desactivar el segundo y activar el tercero), y activar la línea que aplica ambas velocidades ("Aplicar velocidades calculadas").

```
# Resto de asignaciones...

distancia_camara_pelota_inicial =
camara_principal.getVectTo(pelota.position)[0]

def main():
    # Resto de logica...

    distancia_camara_pelota_deseada = distancia_camara_pelota_inicial
    distancia_camara_pelota_actual =
camara_principal.getVectTo(pelota.position)[0]
    velocidad_lineal = 2 * (distancia_camara_pelota_deseada -
distancia_camara_pelota_actual)


camara_principal.setLinearVelocity([0,velocidad_vertical,velocidad_lin
eal],1)
```

Fragmento de código 16. Aplicación de velocidad lineal para mantener constante la distancia entre la cámara y la pelota.

Dado que esta velocidad se aplica directamente sobre el eje Z de la cámara, y éste no es completamente paralelo al plano horizontal (la cámara está inclinada hacia abajo), ocasiona un desplazamiento hacia abajo cuando es negativa y viceversa. Sin embargo esto no es problemático en un principio, ya que en el cuadro siguiente será corregido mediante la aplicación de velocidad vertical.

Si se ejecuta el fichero de ejemplo en este punto, activando el código correspondiente, se comprobará que el funcionamiento del sistema es adecuado, incluso podría ser suficiente para algunos juegos. Todo lo explicado en apartados posteriores estará orientado a considerar la presencia de obstáculos (si se avanza hasta el laberinto se observará que es la principal carencia que presenta), el control mediante el ratón, o a introducir pequeñas mejoras, sin embargo el movimiento básico se seguirá calculando como hasta ahora.

5.4. Giro en el plano horizontal usando el ratón

En el módulo de control de la cámara implementado hasta ahora no ha sido necesario desplazarla lateralmente en ningún momento, solamente hacia adelante y atrás, y hacia arriba y abajo. En este apartado se explica cómo mover la cámara alrededor de la pelota en el plano horizontal mediante el ratón, de manera que el usuario pueda cambiar el punto de vista según desee.

Para seguir los diferentes apartados es conveniente abrir el fichero de prueba ("demo_control_camara.blend") y configurar el controlador Python correspondiente para que lance el módulo "control_camara_2.py" (escribiendo "control_camara_2.main"). En este módulo se han mantenido las funcionalidades del anterior, eliminando las partes superfluas.

5.4.1. Acceso a la posición del ratón y movimiento básico

La referencia al objeto que da acceso al ratón, de modo similar al caso del teclado, se encuentra en el módulo "logic". Por comodidad se almacena en una variable definida en el nivel superior del módulo: `raton = logic.mouse`. Este objeto tiene como atributos dos listas de eventos ("events" y "active_events"), correspondientes al estado de los botones, una lista de dos elementos que representa su posición ("position") y una variable booleana que indica si el puntero es visible durante el juego ("visible").

Para lograr un control natural de la cámara es necesario que sus movimientos vayan acordes con los del ratón, respondiendo a cambios de velocidad o sentido. Hace falta por tanto saber cuánto ha cambiado la posición en cada cuadro respecto al anterior, y aplicar una velocidad a la cámara proporcional a este cambio.

Tras estudiar varias posibilidades se ha llegado a la conclusión de que la manera más práctica y sencilla de conocer el movimiento del ratón en cada momento es la siguiente: en cada cuadro se mide la desviación del puntero respecto al centro de la pantalla, y acto seguido (mediante Python) se vuelve a colocar en el centro. De esta forma se obtiene el vector que uniría la posición anterior del puntero con la actual si ésta no se hubiera modificado artificialmente. Las componentes de este vector son proporcionales a la velocidad del ratón, ya que la diferencia entre ambas posiciones es mayor cuanto más rápido se mueve el ratón.

La velocidad lateral a aplicar se puede obtener (en la versión más básica) simplemente multiplicando la primera componente (horizontal) de este vector por una constante ("SENSIBILIDAD_RATON"). A nivel de código solamente hay que tener en cuenta que el atributo "position" del objeto "ratón" indica la posición de éste respecto a los bordes de la pantalla, siendo la primera componente nula cuando el puntero se encuentra en el borde izquierdo e igual a la unidad si está en el derecho, y lo mismo ocurre con la segunda componente, que es cero en el borde superior y uno en el inferior. Por tanto el centro de la pantalla equivale a la posición [0.5,0.5]. Para distinguir entre posiciones positivas y negativas respecto al punto medio basta con restar 0.5 a cada componente de la posición del ratón.

En el Fragmento de código 17 aparece el código que habría que añadir al desarrollado hasta ahora para implementar una primera aproximación al movimiento lateral controlado mediante el ratón. Su efecto en la práctica se puede comprobar mediante el fichero de ejemplo, activando en el módulo "control_camara_2.py" (si no se ha hecho antes debe configurarse el controlador "Python" para ejecutar la función "main" dicho módulo: "control_camara_2.main") el primer bloque de código que concierne a la velocidad lateral, titulado "Calcular velocidad lateral – movimiento ratón básico".

Nótese cómo la constante "SENSIBILIDAD_RATON" se ha hecho negativa; de esta forma se invierte el sentido en el que se desplaza la cámara. Esta decisión se basa solamente en preferencias personales. En los juegos comerciales suele darse la opción de invertir los ejes de actuación del ratón, cuyo efecto es precisamente este. Por otra parte, en el nivel superior el puntero se configura como visible, para demostrar el principio de funcionamiento del método, mediante el acceso al atributo "visible" del objeto que da acceso al ratón.

```
# Resto de asignaciones...

raton = logic.mouse
raton.visible = True

SENSIBILIDAD_RATON = -500

def main():
    # Resto de logica...

    movimiento_raton = Vector([raton.position[0] -
0.5,raton.position[1] - 0.5])
    raton.position  = (0.5,0.5)
    velocidad_lateral = SENSIBILIDAD_RATON * movimiento_raton[0]

    camara_principal.setLinearVelocity([velocidad_lateral,
velocidad_vertical,velocidad_lineal],1)
```

Fragmento de código 17. Implementación básica del desplazamiento lateral de la cámara gobernado por el ratón.

Al ejecutar el juego se observan algunos problemas que se intentarán solucionar en apartados posteriores:

- Aplicar velocidades laterales muy altas, aparte de otras consecuencias negativas que se explicarán más adelante, ocasiona, si se hace de repente, que la cámara se aleje notablemente de la pelota, ya que la velocidad lateral se aplica perpendicularmente a la dirección cámara-pelota (ver Figura 7, página 47). Esto se puede atajar en cierta medida aumentando la constante que gobierna la aplicación de velocidad lineal, que es la que corrige el alejamiento de la cámara. También ayudará suavizar los movimientos del ratón y evitar cambios bruscos, pero lo más efectivo para evitar éste y otros posibles problemas es limitar la velocidad lateral que es posible alcanzar. Esto se muestra en el Fragmento de código 18, donde se ha limitado al valor de la constante "MAXIMA_VELOCIDAD_LATERAL".
- Al lanzar el juego, el ratón se sitúa por defecto en la posición [0,0], lo que ocasiona que en el primer cuadro, al calcularse la diferencia respecto a [0.5,0.5], se registre un gran movimiento que en realidad es inexistente. Una posible solución sería colocar un contador de cuadros, y descartar el movimiento calculado cuando dicho contador fuese 0. Sin embargo, dado que en el apartado siguiente se va a introducir una lista ("velocidad_lateral_deseada") que se sabe que siempre estará vacía en el primer cuadro, se aprovecha para, cuando lo esté, sustituir el valor presente en "movimiento_raton" por [0,0] (movimiento nulo). En el Fragmento de código 18 se

incluye esta corrección, y en el fichero de ejemplo aparece este mismo bloque de código, llamado "Calcular velocidad lateral – movimiento ratón (problemas coregidos)"; sin embargo, dado que su funcionamiento depende de la lista mencionada, que se introducirá en el apartado siguiente, no es posible activarlo hasta entonces (se indicará convenientemente).

```
# Resto de asignaciones...

MAXIMA_VELOCIDAD_LATERAL = 90

def main():
    # Resto de logica...

    movimiento_raton = Vector([raton.position[0] -
0.5,raton.position[1] - 0.5])
    raton.position  = (0.5,0.5)
    if len(velocidad_lateral_deseada) == 0:
        movimiento_raton = Vector([0,0])

    velocidad_lateral = SENSIBILIDAD_RATON * movimiento_raton[0]
    if fabs(velocidad_lateral) > MAXIMA_VELOCIDAD_LATERAL:
        if velocidad_lateral > 0:
            velocidad_lateral = MAXIMA_VELOCIDAD_LATERAL
        else:
            velocidad_lateral = -MAXIMA_VELOCIDAD_LATERAL

    # Resto de lógica...
```

Fragmento de código 18. Limitación de la velocidad lateral máxima y corrección del valor del movimiento del ratón en el primer cuadro.

- El movimiento no es constante: hay "tirones", aunque se intente mover el ratón con la mayor suavidad posible. Se debe a cómo llegan a Blender los datos del ratón, y probablemente dependa del ordenador y sistema operativo en que se ejecute: si se monitoriza la variable "movimiento_raton" en la consola, se comprueba cómo sus coordenadas cambian constantemente entre valores dispares, a veces con diferencias de más del doble/mitad entre ellos. Se solucionará aplicando un suavizado, explicado en el apartado siguiente.

- En algunas ocasiones, especialmente si el tamaño de la pantalla es pequeño, se observa que la cámara se desplaza ligeramente aunque no se toque el ratón. Esto ocurre cuando el número píxeles que hay físicamente en horizontal en la pantalla es impar: si la pantalla tuviese 25 píxeles, al colocarlo mediante Python en la posición 0.5 debería hacerse teóricamente en el píxel 12.5. Como debe estar sobre un píxel en concreto se pone en el píxel 12, que en proporción al ancho total de la pantalla representa el 12/25 = 48%. Cuando, en el cuadro siguiente, se consultase su posición horizontal, el método devolvería el valor 0.48, y al restar 0.5 para calcular el movimiento, se obtendría 0.02; la cámara se desplazaría acorde con el movimiento que aparentemente ha realizado el ratón, aunque en realidad haya sido nulo.

  Este problema pierde importancia conforme aumenta el tamaño de la pantalla, y sólo se presenta en la mitad de los casos, si se supone que no se establece el tamaño de la

pantalla de forma premeditada. Sin embargo, desaparecerá completamente cuando en apartados posteriores se descarten todos los movimientos del ratón de magnitud menor a una dada (ver apartado 5.6, página 69).

5.4.2. Suavizado de la velocidad lateral

Como se ha comentado, este suavizado es prácticamente imprescindible para obtener una experiencia de usuario decente. Consiste en aplicar la técnica explicada en la primera parte del apartado 7.6 (página 143), es decir, que la velocidad lateral aplicada en cada cuadro sea, en lugar de la que correspondería en función del movimiento del ratón en ese preciso instante, el promedio de las calculadas en un determinado número de cuadros anteriores.

Esto se hace añadiendo en cada cuadro la velocidad lateral calculada a partir del movimiento del ratón ("velocidad_lateral") a una lista tipo "deque" ("velocidad_lateral_deseada"), y calculando la velocidad a aplicar realmente("velocidad_lateral_final") haciendo el promedio de los valores de la lista mediante la función "calcular_media" presente en el módulo "funciones_comunes.py" (explicado en el apartado 7.9.2, página 151), que habrá que importar correctamente en el nivel superior.

En el Fragmento de código 19 aparece el código necesario para conseguir este suavizado. La capacidad de la lista "velocidad_lateral_deseada" no debe ser demasiado grande: con almacenar de 5 a 10 posiciones se ocultan los inconvenientes vistos en el apartado anterior, y un número demasiado elevado introduce un retraso muy notorio en el tiempo de respuesta. Es conveniente experimentar con diferentes valores hasta encontrar el óptimo.

Para probar el efecto del suavizado en el fichero de ejemplo se debe activar el bloque de código titulado "Suavizar velocidad lateral", y modificando, en el bloque "Aplicar velocidades calculadas", la línea activa, para que la primera coordenada del vector pasado a "setLinearVelocity" sea "velocidad_lateral_final" en lugar de "velocidad_lateral". Además, en este punto ya se puede activar el bloque mencionado al final del apartado anterior (llamado "Calcular velocidad lateral – movimiento ratón (problemas corregidos)"), dado que se ha introducido la lista en la cual se apoya ("velocidad_lateral_deseada"); habrá por tanto que desactivar el anterior, "Calcular velocidad lateral – movimiento ratón básico".

```
from funciones_comunes import calcular_media
# Resto de asignaciones...

velocidad_lateral_deseada = deque(maxlen = 10)

def main():
    # Resto de logica...

    velocidad_lateral_deseada.append(velocidad_lateral)
    velocidad_lateral_final =
calcular_media(velocidad_lateral_deseada)

    camara_principal.setLinearVelocity([velocidad_lateral_final,
velocidad_vertical,velocidad_lineal],1)
```

Fragmento de código 19. Suavizado del desplazamiento lateral de la cámara.

Incluso con este suavizado, puede considerarse que la cámara deja de moverse demasiado rápido cuando se para el ratón: por ejemplo, si para hacer un determinado giro no fuera suficiente el margen físico de movimiento del ratón (o la anchura del *touchpad*), y hubiera que dar varias pasadas, entre cada una de ellas la cámara dejaría de moverse, lo cual no es deseable.

Una posible solución consiste en, cuando el valor absoluto de la nueva velocidad lateral es menor que el de la que se aplicó en el cuadro anterior, sustituir la nueva por una fracción de la anterior: por ejemplo, si la cámara tuviera una velocidad lateral de 1, y en un momento dado pasase a ser 0, si la fracción que se aplicase fuera un 0.8 de la anterior, en su lugar se colocaría un 0.8, en el siguiente cuadro un 0.64, y así sucesivamente. Para obtener el valor absoluto se usa la función "fabs" del módulo "math", que es importada al principio del nivel superior.

Esto se debe restringir a los casos donde la nueva velocidad sea bien nula o bien del mismo signo que la anterior. Si los signos son distintos, existe una voluntad del usuario de cambiar el sentido de giro de la cámara, por lo que no se debe aplicar este retraso adicional. En el Fragmento de código 20 se muestra una posible manera de implementarlo (la fracción de la velocidad anterior que se aplica se define mediante la constante "FACTOR_MANTENER_VELOCIDAD_LATERAL"). Es necesario almacenar la velocidad aplicada en cada cuadro en una variable global (ver apartado 3.5 – página 25) de manera que perdure entre cuadros y se pueda recuperar en el siguiente ciclo para hacer la comparación correspondiente. La comprobación de igualdad de signos entre ambas velocidades se hace mediante la función "coinciden_signos", incluida en el fichero "funciones_comunes.py" y explicada en el apartado 7.9.5 (página 153), que habrá que importar en el nivel superior.

En el fichero de ejemplo se puede activar esta característica eliminando las comillas triples del bloque de código titulado "Evitar que la velocidad lateral se pare en seco" y de la línea que almacena la velocidad lateral final en la variable global, situada inmediatamente después del suavizado.

```
from math import fabs
from funciones_comunes import coinciden_signos

# Resto de asignaciones...

velocidad_lateral_anterior = 0

FACTOR_MANTENER_VELOCIDAD_LATERAL = 0.8

def main():
    global velocidad_lateral_anterior

    # Resto de logica...

    if velocidad_lateral == 0 or (fabs(velocidad_lateral) <
fabs(velocidad_lateral_anterior) and
coinciden_signos(velocidad_lateral,velocidad_lateral_anterior)):
        velocidad_lateral = FACTOR_MANTENER_VELOCIDAD_LATERAL *
velocidad_lateral_anterior

    # Resto de logica...

    velocidad_lateral_anterior = velocidad_lateral_final
```

Fragmento de código 20. Suavizado específico de la detención de la velocidad lateral.

5.5. Control de choques en el plano horizontal

Al ejecutar el juego, mientras que el seguimiento de la pelota y el control de la cámara mediante el ratón son en general satisfactorios, cuando hay una gran cantidad de obstáculos se echa en falta un mecanismo que impida que la cámara atraviese las paredes (haciendo que la pelota se pierda de vista). Como se comentó en el apartado 5.2.1 (página 47), no es posible aprovechar que la cámara es un sólido rígido para evitarlo, ya que de esa manera, además de que ocasionaría cambios al chocar con objetos móviles, se quedaría detrás de los obstáculos cuando chocase de frente con ellos (por ejemplo al doblar una esquina). En este apartado se explica cómo gestionar los choques mediante Python para evitar perder de vista la pelota.

El fichero de ejemplo se debe configurar para que el controlador "Python" correspondiente ejecute la función "main" del módulo "control_camara_3.py", que incluye el código correspondiente a este apartado.

La lógica necesaria se divide en dos ámbitos diferenciados, los relacionados con el desplazamiento lateral y lineal, cuyos razonamientos lógicos son los siguientes:

- Desplazamiento lateral: la cámara debe dejar de responder a la velocidad lateral cuando se den, al mismo tiempo, las siguientes condiciones:
    - La cámara esté cerca de una pared, lo cual se detecta mediante un sensor "Near".
    - La pelota esté visible. Si la pelota no se ve, hay un obstáculo entre la cámara y ella, y por tanto sí se debe permitir que lo atraviese. La visibilidad se determina usando el método "rayCastTo", presente en todos los objetos.

  - La velocidad lateral tienda a llevarla hacia la pared. Si después del choque (de la detección) se pretende aplicar una velocidad lateral en el sentido contrario al que llevaba cuando chocó, sí se debe permitir.

- Desplazamiento lineal: si la cámara está avanzando hacia atrás, y se acerca a una pared, debe dejar de hacerlo para no atravesarla y de esa forma perder de vista la pelota. El choque/cercanía se detecta mediante un sensor "Near" (diferente al dedicado al desplazamiento lateral, como se explicará a continuación).

5.5.1. Configuración y uso de los sensores "Near"

Un sensor "Near" es un tipo especial de sensor "Touch", que se activa cuando hay un objeto dentro de una esfera de detección con un radio determinado, en lugar de cuando se llega a producir contacto (radio nulo). La razón por la que se han usado sensores "Near" en lugar de "Touch" es precisamente esta característica, que permite establecer distintos márgenes de detección en distintos sensores.

Por cada sensor es necesario añadir el *logic brick* correspondiente al objeto "camara_principal", darle un nombre, dejando el resto de los parámetros sin modificar, y conectarlo a un controlador cualquiera (es indiferente, solo se conectan para que Blender los evalúe en cada cuadro, mientras que el acceso se hará desde Python).

En Python se debe, en el nivel superior, conseguir una referencia al sensor (buscando en la lista "sensors" del objeto "camara_principal" por el nombre que se dio al *logic brick*), y configurarlo adecuadamente. En este caso sólo será necesario configurar el radio de la esfera de detección (atributo "distance") y la distancia a la cual el sensor deja de estar activo (atributo "resetDistance"), que en este caso será igual a la de detección.

En la función que controla la lógica en cada cuadro, en este caso "main", sólo es necesario consultar el estado del sensor y actuar en consecuencia. Como solamente hace falta distinguir si hay algún objeto dentro de la esfera de detección o no, basta con acceder al atributo "positive".

En el Fragmento de código 21 aparece un ejemplo de cómo configurar y usar un hipotético sensor "Near". No se hace uso de posibilidades más avanzadas, como distinguir entre materiales, propiedades, o diferenciar entre distintos estados (recién activado, recién desactivado, etc.), porque no son necesarias para este propósito. Nótese cómo la variable "pared_cerca" se ha definido únicamente para aumentar la legibilidad del código, ya que es completamente prescindible.

```
# Resto de asignaciones...

sensor_camara_cerca = camara_principal.sensors['cerca']
sensor_camara_cerca.distance = 1
sensor_camara_cerca.resetDistance = 1

def main():
    pared_cerca = sensor_camara_cerca.positive

    if pared_cerca:
        # Acciones correspondientes...
```

Fragmento de código 21. Configuración y uso básico de un sensor “Near”.

5.5.2. Uso del método “rayCastTo”

La función que desempeña este método podría llevarse a cabo igualmente mediante un sensor “Ray”; sin embargo, dado que el uso de este último es algo más complejo, y que en este proyecto se busca usar Python en lugar de *logic bricks* siempre que sea posible, se ha optado por “rayCastTo”.

El método “rayCastTo” está presente en todos los objetos del juego (objetos de clase “KX_GameObject”). Su funcionamiento es el siguiente: devuelve el primer objeto que corta el rayo que va desde el objeto que lo llama hasta otro dado (o hasta un punto cualquiera del espacio). El único parámetro obligatorio es el objeto o punto hacia el que lanzar el rayo. Mediante otros dos se puede restringir el resultado a objetos que tengan una determinada propiedad o que estén a una distancia dada, pero en este caso no es conveniente hacerlo.

Su uso en esta ocasión será el siguiente: en cada cuadro, se lanza un rayo desde la cámara hasta la pelota. Si el objeto devuelto es la pelota, ésta es visible. De lo contrario hay al menos un obstáculo entre ambas. La línea de código a ejecutar en cada cuadro es: `hay_obstaculo = False if camara_principal.rayCastTo(pelota) == pelota else True` (usando una expresión condicional, que permite escribir una estructura if/else en una sola línea). Tras ella, la variable “hay_obstaculo” será cierta cuando la pelota no sea visible, lo que permitirá tomar las decisiones oportunas.

5.5.3. Choques causados por un desplazamiento lateral

Como se comentó anteriormente, una vez se detecte un choque (en realidad se detecta la proximidad de otro objeto), si la pelota era visible en el momento de la detección, se debe impedir que se aplique a la cámara una velocidad lateral del mismo signo que la que llevaba antes de chocar, de manera que no atraviese el objeto contra el que ha chocado. Por el contrario, si la pelota no era visible se debe permitir que la cámara atraviese la pared, ya que el objetivo del desplazamiento lateral será recuperar la visibilidad.

En el módulo correspondiente (en este caso “control_camara_3.py”), la lógica necesaria en cada cuadro se distribuye en tres secciones (ver Fragmento de código 22):

- En primer lugar se interpretan el estado del sensor "Near" (llamado "sensor_camara_cerca_lateral") y el resultado del método "rayCastTo": si el sensor "Near" es positivo, es decir hay otro objeto cerca (en este caso sólo puede ser una pared), la variable "pared_cerca_lateral" es verdadera, y viceversa; si el método "rayCastTo" devuelve un objeto distinto de la pelota, la variable "hay_obstaculo" será verdadera, en caso contrario falsa.
- A continuación se deduce, con base en lo anterior, si es necesario bloquear la velocidad lateral, y en caso afirmativo, qué sentido (sólo se debe evitar que la cámara se siga desplazando hacia la pared). La información del resultado de este bloque se guarda en la variable "parar_velocidad_lateral", que indica si hay que evitar las velocidades laterales negativas (cuando su valor es -1), positivas (1) o si no hay que hacerlo, en función de la velocidad lateral de la cámara en el cuadro anterior. Esta variable debe perdurar entre cuadros: mientras la cámara esté cerca de una pared deberá seguir evitándose que se acerque a ella, sin embargo una vez que se ha parado no podría saberse, si no perdurase, la velocidad que llevaba antes de chocar ("velocidad_lateral_anterior" sería nula); por tanto "parar_velocidad_lateral" debe mantener el mismo valor mientras el sensor "Near" esté activo. Los pasos que se dan son los siguientes:
    - Si se detecta la cercanía de otro objeto, y la pelota está visible, la variable "parar_velocidad_lateral" será 1 o -1, y habrá que deducir su valor. Si no, será nula.
    - En caso de que se haya detectado, si la variable "parar_velocidad_lateral" era nula anteriormente (el actual es el primer cuadro en que se detecta el choque) habrá que calcular su valor; si no, se mantiene el anterior.
    - En caso de que fuera nula, su nuevo valor será -1 si la velocidad lateral que llevaba la cámara anteriormente "velocidad_lateral_anterior" era negativa, 1 en caso contrario. Esta lógica de decisión es adecuada para conseguir un comportamiento correcto en la mayoría de los casos, pero fallará en situaciones donde la cámara esté cerca de una pared y no se haya movido el ratón con anterioridad, ya que "velocidad_lateral_anterior" será nula. La solución es tomar la decisión en función del sentido de giro sobre sí misma de la cámara (la cámara gira tanto cuando se mueve la pelota como cuando se mueve ella misma). El cálculo del sentido de giro se cubre en el apartado 5.6.2 (página 71), por lo que a partir de entonces esta decisión se tomará basándose en el mismo.
    - Si se cumplen las condiciones anteriores pero la cámara acaba de atravesar una pared (recordar que se permite si la pelota no es visible), el signo de la velocidad a evitar no será el que llevaba en el cuadro anterior, si no el contrario. Para saber si se da esta situación hay que comprobar la variable "pelota_visible_cuadro_anterior", que es global y se define al final de este

bloque lógico como el contrario de “hay_obstaculo”; si es falsa efectivamente se acaba de atravesar una pared, y por tanto habrá que invertir el valor de “parar_velocidad_lateral”.

- Posteriormente se ejecuta el resto de la lógica con normalidad, calculado todas las velocidades correspondientes en cada cuadro. Aunque en este caso el orden de estos dos primeros bloques podría haberse invertido, posteriormente se añadirán funcionalidades que dependan del estado de “parar_velocidad_lateral”, por lo que debe definirse previamente.
- Por último, justo antes de aplicar finalmente las velocidades, es necesario determinar si la lateral se debe anular o no: si “parar_velocidad_lateral” no es nula, y la velocidad lateral que se ha calculado en función del movimiento del ratón tiene su mismo signo, así deberá hacerse, pasando a ser “velocidad_lateral_final” igual a cero. La igualdad de signos se comprueba mediante la función “coinciden_signos”, (apartado 7.9.5, página 153)

En el Fragmento de código 22 aparece toda la lógica relacionada con lo anterior. Se recomienda observarlo con atención para entender su funcionamiento, siguiendo los caminos que se siguen en función de diferentes situaciones. Para probar el comportamiento en el fichero de ejemplo se deben activar tanto la primera parte de la función “main” del módulo “control_camara_3.py”, que contiene la interpretación de los sensores, del método “rayCast” y la decisión del sentido de giro a evitar, como el bloque que decide si finalmente se debe anular la velocidad lateral que se ha calculado, titulado “Parar velocidad lateral”.

```
# Resto de asignaciones...

parar_velocidad_lateral = 0
pelota_visible_cuadro_anterior = 0

sensor_camara_cerca_lateral =
camara_principal.sensors['cerca_lateral']
sensor_camara_cerca_lateral.distance = 1
sensor_camara_cerca_lateral.resetDistance = 1

def main():
    global velocidad_lateral_anterior
    global parar_velocidad_lateral
    global pelota_visible_cuadro_anterior

    # Interpretar estado del sensor y resultado del método
    pared_cerca_lateral = sensor_camara_cerca_lateral.positive
    hay_obstaculo = False if camara_principal.rayCastTo(pelota) ==
pelota else True

    # Deducir qué sentido de giro se debe evitar
    if pared_cerca_lateral and not hay_obstaculo:
        if parar_velocidad_lateral == 0:
            parar_velocidad_lateral = 1 if velocidad_lateral_anterior
> 0 else -1
            if not pelota_visible_cuadro_anterior:
                parar_velocidad_lateral = - parar_velocidad_lateral
    else:
        parar_velocidad_lateral = 0
    pelota_visible_cuadro_anterior = 1 - hay_obstaculo

    # Resto de lógica...

    # Decidir si la velocidad lateral calculada se debe aplicar o no
    if parar_velocidad_lateral:
        if coinciden_signos(parar_velocidad_lateral,
velocidad_lateral_final):
            velocidad_lateral_final = 0

    # Resto de lógica...
```

Fragmento de código 22. Tratamiento de los choques causados por un desplazamiento lateral.

5.5.4. Choques causados por un desplazamiento lineal

El desplazamiento lineal de la cámara (en la dirección cámara-pelota) también debe tenerse en cuenta a la hora de evitar que la cámara se oculte tras las paredes. Concretamente, se debe evitar que esto ocurra cuando la pelota está avanzando “hacia atrás”, y por tanto la cámara está haciendo lo mismo para mantener la distancia entre ambas.

La solución es más simple en este caso: si se detecta la cercanía de una pared, y la velocidad lineal que se va a aplicar a la cámara es positiva (es decir, tendería a separarse de la pelota), esa velocidad se hace nula.

Solamente hay que tener en cuenta que es necesario usar un sensor “Near” distinto al del apartado anterior. Si se usase el mismo, su activación ocasionaría el bloqueo de ambas velocidades, lo cual no tiene por qué ser necesario: por ejemplo, si la cámara se está moviendo hacia atrás perpendicularmente a una pared debe dejar de hacerlo cuando se detecte

la cercanía; sin embargo, debería poder moverse hacia los lados, ya que ello no implica que se coloque detrás de la pared. Si este movimiento lateral sí que es problemático (por ejemplo si se acerca a la pared en diagonal), será detenido cuando se active el sensor correspondiente.

Se tienen por tanto dos sensores "Near", uno dedicado a la velocidad lateral ("sensor_camara_cerca_lateral") y otro a la lineal ("sensor_camara_cerca_lineal"). Para que se comporten adecuadamente el radio de detección del correspondiente a la velocidad lineal debe ser mayor que el correspondiente a la lateral. Si no fuera así el correspondiente a la velocidad lateral se activaría en todos los casos, con lo cual no se cumpliría lo explicado en el párrafo anterior.

La lógica necesaria aparece en el Fragmento de código 23. Para usarlo en el fichero de ejemplo basta con activar las dos líneas de código que determinan si se debe parar la velocidad lineal, situadas al final del bloque "Calcular velocidad lineal".

```
# Resto de asignaciones...

sensor_camara_cerca_lineal = camara_principal.sensors['cerca_lineal']
sensor_camara_cerca_lineal.distance = 1.5
sensor_camara_cerca_lineal.resetDistance = 1.5

def main():
    pared_cerca_lineal = sensor_camara_cerca_lineal.positive

    # Resto de lógica...

    velocidad_lineal = 2 * (distancia_camara_pelota_deseada -
distancia_camara_pelota_actual)
    if pared_cerca_lineal and velocidad_lineal > 0:
        velocidad_lineal = 0

    # Resto de lógica...
```

Fragmento de código 23. Tratamiento de los choques causados por un desplazamiento lineal.

5.5.5. Consideraciones

Al ejecutar el fichero habiendo activado todos los bloques de código correspondientes, se observará que el comportamiento se ajusta a lo esperado, excepto por los siguientes detalles:

Por una parte, si se mueve el ratón rápidamente y la cámara choca contra una pared, probablemente la atraviese. Esto probablemente ocurra porque, al ser la velocidad lateral alta, la cámara recorre una distancia grande en cada cuadro; si esta distancia recorrida en cada cuadro es mayor que el radio de detección de los sensores "Near", es fácil que los sensores no se evalúen mientras la cámara está dentro del radio. Es decir, si se mueve rápidamente puede ocurrir que en un ciclo lógico se encuentre delante de una pared pero sin que se detecte el contacto, y en el siguiente ya la haya atravesado.

Este problema puede minimizarse reduciendo la velocidad máxima lateral de la cámara ("MAXIMA_VELOCIDAD_LATERAL"), aumentando el radio de la esfera de detección del sensor "Near" (teniendo en cuenta que no debe detectar el suelo, o si lo hiciera excluyendo los

casos en que es el único objeto detectado, operando con "sensor_camara_cerca_lateral.hitObjectList") o con una combinación de ambos.

Por otra parte, al poner a prueba expresamente el control de los choques por velocidad lineal, avanzando hacia atrás hacia una pared, se observará, una vez la cámara deja de retroceder, que no está completamente parada, si no que se desplaza ligeramente hacia adelante.

Esto es una consecuencia de aplicar directamente las velocidades sobre los ejes locales de la cámara: como es necesario mantener la posición vertical de la cámara aplicando una cierta velocidad vertical, y ello se hace directamente sobre el eje Y de la cámara, que no apunta exactamente en vertical si no que está inclinado ligeramente hacia adelante, esta velocidad ocasionará también un desplazamiento lineal. Normalmente estas dos magnitudes se compensan automáticamente, pero en este caso no es posible ya que expresamente se anulan las velocidades lineales mayores que 0 (como lo es la necesaria para compensar).

La solución es aplicar la velocidad a la cámara usando coordenadas globales, tomando como referencia la dirección cámara-pelota, de manera que los movimientos en un eje no influyan en el resto. No se ha adoptado desde el principio porque los inconvenientes de no hacerlo se pueden considerar irrelevantes, y se pretende usar la menor cantidad de código posible para hacerlo menos complicado. Sin embargo, una vez se introduzca el movimiento en vertical usando el ratón los problemas generados por el uso de coordenadas serán más importantes, y será entonces cuando se cambie a las globales. Este cambio se explica en el apartado 5.8.4 (página 88).

Por último, al hacer que la cámara choque con determinados elementos del escenario, especialmente cuando hay dos paredes muy próximas, el sistema falla, no evitando que las atraviese. Estos objetos se han colocado especialmente para poner a prueba la lógica que se implementará cuando se añada el movimiento vertical, y por tanto será en el apartado correspondiente (5.9, página 91) cuando se refine el código para tener en cuenta estos casos.

5.6. Giro automático en el plano horizontal para evitar obstáculos

En general, los juegos están diseñados para poder guiar al personaje por la escena sin necesidad de usar el ratón para ello. En este caso, aunque la cámara sigue correctamente a la pelota, se hace necesario corregir su posición con el ratón cuando se interponen obstáculos entre ambas, especialmente al doblar esquinas, como se puede comprobar en el laberinto del ejemplo.

En este apartado se va a explicar el funcionamiento del sistema que se ha implementado para, cuando se detecte un obstáculo entre la cámara y la pelota, la primera haga un giro en el plano horizontal para evitarlo. En el fichero de ejemplo se ha incluido el módulo "control_camara_4.py", que contiene toda la lógica tratada; para usarlo se debe configurar el controlador "Python" correspondiente al control de la cámara para que lance su función "main" ("control_camara_4.main").

5.6.1. Activación

El primer aspecto que se debe tener en cuenta es cuándo se debe efectuar este giro automático. Dado que controla una función que también puede realizar el usuario moviendo el ratón, lo lógico es que lo haga cuando el ratón está parado; es decir, se da prioridad a la voluntad del usuario, aunque ello suponga perder de vista la pelota. Para discernir entre cuándo el control se está efectuando manualmente y cuando no, no basta con tener en cuenta solamente el valor del movimiento del ratón en cada cuadro (nulo/no nulo); se deben considerar dos aspectos:

- Como se explicó en el apartado 5.4.1 (página 57), cuando el número de píxeles de la pantalla es impar se interpreta erróneamente que el ratón está continuamente en movimiento, aunque esté parado. Para solucionarlo, se descartan los movimientos en los que el puntero se desplace en horizontal menos de un píxel (el error que se produce es de medio píxel). Estos no existirán en la realidad, con lo cual se puede corregir el error sin perder sensibilidad. Para calcular la magnitud que resultaría de un movimiento del ratón de un píxel ("margen_error_raton") se hace el inverso de la anchura de la pantalla en píxeles, que se obtiene mediante el método "getWindowWidth" del módulo "render". En el Fragmento de código 24 aparece la función que calcula el movimiento del ratón modificada para incluir esta discriminación.
- Aún con la corrección anterior, no es práctico activar el sistema automático simplemente cuando el movimiento del ratón sea nulo. Si se monitoriza la variable "movimiento_raton", se comprobará que al moverlo lentamente el valor obtenido alterna continuamente entre magnitudes pequeñas y "False". Esto resultaría en que el movimiento manual y el automático se alternarían entre cuadros, de manera que ninguno conseguiría su objetivo, y además resultaría molesto para el usuario. La manera más simple de solucionarlo es activar el sistema automático sólo cuando haya pasado un determinado número de cuadros desde la última vez que se detectó movimiento en el ratón, lo que se puede conseguir usando una variable global ("cuadros_sin_raton") que, en cada cuadro, dependiendo de si el ratón se ha movido o no, aumente su valor o se haga nula. En el Fragmento de código 24 se ha incluido este razonamiento.

Por tanto, la cámara girará automáticamente cuando un obstáculo se interponga en su línea de visión, siempre que el usuario lleve un determinado número de cuadros sin mover el ratón. Si se registran movimientos indeseados (por ejemplo si el ratón es muy sensible y detecta cambios aun cuando se pretende mantener quieto) se puede establecer un margen de error más alto manualmente, aunque en este caso sí sería a costa de perder algo de precisión.

En el Fragmento de código 24 se ha incluido tanto la consideración del margen de error del ratón como el mecanismo que cuenta los cuadros durante los que no se ha movido, y en el módulo "control_camara_4.py" todo el bloque aparece activado por defecto, no siendo necesario hacer ningún cambio por el momento.

```
from bge import logic, render
# Resto de asignaciones...

cuadros_sin_raton = 0
margen_error_raton = 1 / render.getWindowWidth()

def main():
    global cuadros_sin_raton

    # Resto de lógica...

    movimiento_raton = Vector([raton.position[0] -
0.5,raton.position[1] - 0.5])
    raton.position  = (0.5,0.5)
    if fabs(movimiento_raton[0]) < margen_error_raton:
        movimiento_raton = False
    if len(velocidad_lateral_deseada) == 0:
        movimiento_raton = False
    if movimiento_raton:
        velocidad_lateral = SENSIBILIDAD_RATON * movimiento_raton[0]
        cuadros_sin_raton = 0
    else:
        velocidad_lateral = 0
        cuadros_sin_raton += 1
```

Fragmento de código 24. Modificaciones introducidas al cálculo del movimiento del ratón.

5.6.2. Cálculo del sentido de giro de la cámara

Una vez se conoce cuándo se debe girar la cámara automáticamente, es imprescindible saber en qué sentido hay que hacerlo. Por ejemplo, si el objeto protagonista dobla una esquina hacia la izquierda, lo natural es que la cámara se desplace hacia la derecha para volver a tenerlo en su campo de visión (si lo hiciera hacia la izquierda necesitaría desplazarse más, atravesar las paredes, y cuando volviera a "ver" al objeto, éste probablemente estaría avanzando hacia ella).

Esto se traduce en hacer que la cámara se desplace acorde con el sentido en el que estaba girando sobre sí misma antes de aparecer el obstáculo. Debido a las bases del movimiento de la cámara (desplazamiento + rotación), siempre que la pelota se mueva lateralmente, lo cual es necesario para que se coloque detrás de un obstáculo, la cámara estará girando sobre sí misma en el plano horizontal (ver apartado 5.2.3, página 49). Siguiendo con el ejemplo anterior, si la pelota se está moviendo hacia la izquierda para doblar una esquina, la cámara estará girando hacia la izquierda para mantenerla en el centro de su campo de visión. Una vez se interponga la esquina, la cámara se deberá desplazar hacia la derecha.

Por tanto, es necesario saber en qué sentido rota la cámara en el plano horizontal; una posible manera es la siguiente: se tiene en cuenta la dirección del plano horizontal en la que apunta la cámara, y la dirección en la que lo hacía en el cuadro anterior. Se calculan los ángulos que forman ambas direcciones con el eje X (siempre del plano horizontal); si el ángulo actual es mayor que el anterior, estará rotando hacia la izquierda, si no hacia la derecha.

Las direcciones de apunte se obtienen de la matriz "worldOrientation" de la cámara (ver apartado 7.8, página 147). Se entiende que la dirección en la que apunta coincide con la parte negativa de su eje Z, por tanto habrá que tomar los valores de la tercera columna y cambiarlos de signo. Como sólo se tiene en cuenta el plano horizontal, el tercer valor se descarta, resultando un vector bidimensional.

Para calcular el ángulo formado entre un vector y el eje X se usa la función "obtener_angulo_x" del módulo "funciones_comunes.py" (explicada en el apartado 7.9.3, página 152) que habrá que importar en el nivel superior. Esta función devuelve ángulos positivos expresados en radianes, entre 0 y 2*pi.

La diferencia entre los ángulos formados con el eje X de ambas direcciones se puede relacionar directamente con el sentido de giro de la cámara en todos los casos, con una excepción: cuando una de las direcciones está a un lado del eje X y la otra en el contrario, la diferencia resultará de signo opuesto al real. Esto se corrige comprobando si el valor absoluto de la diferencia es mayor que pi, en cuyo caso habrá que invertir el sentido calculado.

Una vez calculado el sentido de giro es necesario almacenar el ángulo actual de la cámara en la variable global "angulo_camara_anterior", para poder usarlo en el siguiente cuadro.

En el Fragmento de código 25 se refleja el código correspondiente al cálculo del sentido de giro de la cámara en cada cuadro. Este código se encuentra también activado por defecto en el módulo "control_camara_4.py" (el bloque se denomina "Calcular el sentido de giro de la cámara")

```
from funciones_comunes import obtener_angulo_x

# Resto de asignaciones...

angulo_camara_anterior = 0

def main():
    # Resto de logica...

    angulo_camara_actual = obtener_angulo_x([-
camara_principal.worldOrientation[0][2],-
camara_principal.worldOrientation[1][2]])
    if angulo_camara_actual > angulo_camara_anterior:
        sentido_giro_camara = 1
    else:
        sentido_giro_camara = -1
    if fabs(angulo_camara_actual - angulo_camara_anterior) > pi:
        sentido_giro_camara = -sentido_giro_camara

    angulo_camara_anterior = angulo_camara_actual
```

Fragmento de código 25. Cálculo del sentido de giro de la cámara.

5.6.3. Cálculo de la velocidad lateral correspondiente

Una vez se dispone de los datos necesarios, se deben interpretar y usar para calcular, cuando sea necesario, la velocidad lateral a aplicar a la cámara para evitar los obstáculos. En este caso el razonamiento es simple: si el ratón lleva sin moverse más de un determinado

número de cuadros (por ejemplo 100), se comprueba si hay algún obstáculo entre la cámara y la pelota. De ser así, se calcula la velocidad lateral ("velocidad_lateral") multiplicando el sentido de giro de la cámara obtenido anteriormente por una constante que determinará la rapidez del giro.

El bloque de código correspondiente debe colocarse después del cálculo del movimiento del ratón, para que la variable "cuadros_sin_raton" esté actualizada, después de comprobar si hay obstáculos, y antes del resto de bloques que modifican "velocidad_lateral" o calculan el valor final a aplicar a partir de ella.

Existe la opción de incluir el cálculo del sentido de giro de la cámara dentro de este bloque, de manera que sólo se ejecute cuando ésta deba evitar un obstáculo. Si así se hiciese habría que recordar almacenar la dirección de apunte de la cámara en cada cuadro para poder usarla en el siguiente si fuera necesario. En este caso se han mantenido como bloques independientes para conocer en qué sentido gira la cámara aunque no haya obstáculos ya que de esta manera es posible usar la información para tomar decisiones en otros puntos del módulo. Por ejemplo, como se comentó en el apartado correspondiente, el control de choques contra las paredes se basará a partir de ahora en el sentido de giro de la cámara cuando se produce el choque.

En el Fragmento de código 26 aparece el código correspondiente al cálculo de la velocidad cuando se dan las condiciones necesarias. En el módulo "control_camara_4.py" del fichero de ejemplo basta con activar el bloque correspondiente ("Giro automático en el plano horizontal para evitar obstáculos"); asimismo, para hacer que la detección de choques se base en el sentido de giro de la cámara, se debe, en dicho bloque ("Deducir qué sentido de giro se debe parar"), activar la asignación que se encuentra entre comillas triples y anular la opuesta.

```
# Resto de asignaciones...

FACTOR_GIRO_EVITAR_OBSTACULO = 5

def main():
    # Resto de lógica...

    if cuadros_sin_raton > 100:
        if hay_obstaculo:
            velocidad_lateral = FACTOR_GIRO_EVITAR_OBSTACULO *
sentido_giro_camara
```

Fragmento de código 26. Cálculo de la velocidad lateral a aplicar para evitar obstáculos.

Nótese que es posible hacer la multiplicación directamente porque la variable "sentido_giro_camara", se preparó para que su valor, multiplicado por una constante, produjera el desplazamiento adecuado al aplicarlo a la componente X en "setLinearVelocity". Por ejemplo, cuando la cámara está girando hacia la izquierda (el ángulo actual con el eje X es mayor que el anterior), "sentido_giro_camara" toma el valor 1; al multiplicarlo por la

constante resulta un valor positivo, que aplicado como velocidad al eje X provoca un desplazamiento hacia la derecha.

5.7. Orientación de la cámara en la dirección de desplazamiento del objeto protagonista

Aunque, dada la naturaleza del método que se ha usado para efectuar el seguimiento de la cámara al objeto protagonista, la primera siempre tiende a apuntar en su dirección de avance cuando lo hace hacia adelante, no es así cuando lo hace lateralmente o hacia atrás (como si el objetivo estuviese unido a la pelota mediante un hilo). El problema más claro que esto genera, es decir, que se interpongan obstáculos entre ambas, se solucionó en el apartado anterior, sin embargo pueden darse situaciones donde sea necesario o conveniente orientar la cámara expresamente en la dirección en la que avanza el objeto protagonista.

En este apartado se va explicar la forma que se ha considerado más simple de conseguirlo. Dado que la manera en que se use esta funcionalidad dependerá de lo necesario en cada caso, en primer lugar se explicará cómo efectuar el giro de la cámara propiamente dicho, y posteriormente se expondrán diferentes lógicas de activación (cuándo hacerlo). Todo lo referente a este apartado se refleja en el módulo "control_camara_5.py" del fichero de ejemplo ("demo_control_camara.blend"), por lo que se debe configurar el controlador "Python" correspondiente para que lance su función "main" ("control_camara_5.main").

5.7.1. Cálculo del sentido y la magnitud de la velocidad lateral

El método usado para realizar el giro es de nuevo la aplicación de un desplazamiento lateral a la cámara (mediante la primera componente del vector aplicado al método "setLinearVelocity" en modo local) que, combinado con la rotación de la misma que se realiza en cada cuadro (mediante "alignAxisToVect"), ocasiona un giro alrededor de la pelota. En este caso, al contrario que al evitar obstáculos, la elección del sentido de la velocidad lateral no debe basarse en cómo rotaba la cámara anteriormente, sino que se elegirá el que conlleve realizar un giro menor hasta alcanzar la nueva orientación (es decir, como mucho se girarán 180º).

Las direcciones a considerar son la de apunte de la cámara (el sentido negativo de su eje Z), que se obtiene de su matriz "worldOrientation", y la de avance de la pelota, presente en la propiedad "direccion_pelota" del objeto "pelota", cuyo cálculo se explica en el apartado 7.7 (página 144). Una vez más es necesario disponer de los ángulos que éstas forman respecto a una referencia (en este caso el eje X), para poder operar con ellas. El formado por la dirección de apunte de la cámara ya se obtuvo en el apartado anterior ("angulo_camara_actual"), al deducir su sentido de giro, y el correspondiente a la pelota ("angulo_direccion_pelota") se calcula de nuevo mediante la función "obtener_angulo_x", del módulo "funciones_comunes.py" (apartado 7.9.3, página 152).

Para conocer en qué sentido se debe realizar el giro para que éste sea de menos de 180º, lo más directo es usar la función "obtener_angulo_diferencia". Esta función, cuyo

funcionamiento se explica en el apartado 7.9.4 (página 153), devuelve el ángulo (con signo) más pequeño entre dos dados, que se pasan como parámetros. El segundo de estos parámetros se considera el ángulo de referencia, de manera que se devuelve el ángulo, entre –pi y pi radianes que el primero forma con él.

En este caso, el valor devuelto por esta función se ha aprovechado doblemente de manera directa para calcular la velocidad lateral:

- El sentido se relaciona con el signo del ángulo devuelto. Supóngase una situación donde la pelota avanza en la dirección [1,0] (0º) y la cámara apunta hacia [0,1] (90º), y por tanto esta última debe girar hacia la derecha (aplicando una velocidad lateral hacia la izquierda) para que sus direcciones coincidan. Si se toma como referencia el ángulo que forma la dirección de la pelota, la llamada a la función quedaría: `obtener_angulo_diferencia(angulo_camara_actual,angulo_direccion_pelota)`, y el resultado serían pi/2 radianes (90º). Dado que se necesita que la cámara se desplace hacia la izquierda (velocidad lateral negativa), se deduce que el sentido necesario es siempre el opuesto al definido por el signo del ángulo devuelto.
- La magnitud se puede definir como el valor absoluto del ángulo devuelto por "obtener_angulo_diferencia" multiplicado por una constante (FACTOR_GIRO_ORIENTAR_CAMARA) que determinará la rapidez. De esta manera, como el ángulo es menor conforme se va realizando el giro, la velocidad aplicada se irá reduciendo, consiguiéndose un suavizado directamente. También podría definirse simplemente como una constante, o combinando ambos métodos, lo que ofrecería un control más fino sobre el comportamiento.

Dado que tanto la magnitud como el sentido de la velocidad dependen directamente de la magnitud y el signo del valor devuelto por la función (denominado "diferencia_angulos" en los ejemplos), la asignación se puede hacer directamente: `velocidad_lateral = -diferencia_angulos * FACTOR_GIRO_ORIENTAR_CAMARA` (Nótese que el signo de la velocidad lateral debe ser el opuesto al del ángulo diferencia).

En el Fragmento de código 27 aparece el bloque de código más básico que habría que añadir a lo anterior para que la cámara se orientase en la dirección de la pelota. En este caso el sistema se activa simplemente al mantener pulsada la tecla I (el acceso al teclado se explica en el apartado 7.1, página 133). En el módulo "control_camara_5.py" el bloque de código (llamado "Giro automático en el plano horizontal para orientar la cámara en la dirección de la pelota") se ha colocado justo después del correspondiente al giro para evitar obstáculos, y está activo por defecto. Tanto si se hace después como si se hace antes existirán conflictos entre los sistemas, que se tratan en el siguiente apartado.

```
from bge import logic, render, events
# Resto de asignaciones...

FACTOR_GIRO_ORIENTAR_CAMARA = 10

def main():
    # Resto de lógica...

    if teclado.events[events.IKEY] == 2:
        angulo_direccion_pelota =
obtener_angulo_x(pelota['direccion_pelota'])
        diferencia_angulos =
obtener_angulo_diferencia(angulo_camara_actual,angulo_direccion_pelota
)
        velocidad_lateral = -diferencia_angulos *
FACTOR_GIRO_ORIENTAR_CAMARA
```

Fragmento de código 27. Orientación de la cámara en la dirección de avance de la pelota al pulsar una tecla.

5.7.2. Tratamiento de obstáculos

Como se puede comprobar ejecutando el juego, al mantener pulsada la tecla I la cámara se orienta correctamente en la dirección de la pelota, y se detiene en caso de que choque contra alguna pared. Sin embargo, como el código se ha colocado después del dedicado a evitar obstáculos en la línea de visión, su efecto quedará anulado al re-definirse la variable "velocidad_lateral", y la cámara se detendrá cuando alcance la dirección objetivo aunque la pelota no sea visible. Esto se puede comprobar colocando la pelota a un lado de la pared que aparece a la izquierda cuando se lanza el juego, haciéndola avanzar ligeramente en dirección contraria a la pared y parándola; si acto seguido se pulsa la tecla I, la cámara se colocará al otro lado de la pared, mirando hacia la pelota. Si se suelta dicha tecla se corregirá la situación, pero en cuanto se pulse de nuevo volverá a ocurrir.

Si el código correspondiente al giro para evitar obstáculos se hubiera colocado después, la cámara no pararía detrás del obstáculo, pero una vez que éste hubiera sido salvado y por tanto no actuase el sistema correspondiente, se volvería a mover hacia la dirección deseada, lo que provocaría que el obstáculo se interpusiera de nuevo, y así sucesivamente.

La solución más efectiva de las estudiadas es la siguiente: permitir que la cámara gire sólo cuando la dirección a adoptar no implique colocarse detrás de un obstáculo. Para ello se predice la posición que adoptaría la cámara una vez realizado el giro, y se comprueba si entre dicha posición y la que ocupa la pelota hay obstáculos.

Para calcular la posición que ocuparía la cámara basta con sumar, a las coordenadas horizontales de la pelota, el vector que se forma al multiplicar la distancia inicial entre la cámara y la pelota (la que se tiende a mantener en todo momento) por la dirección opuesta a la de desplazamiento de la pelota. Es decir, esta posición es el punto final de un vector con origen en la pelota, dirección igual a la de la pelota, sentido contrario, y longitud la distancia entre la cámara y la pelota. Como los vectores que se manejan son bidimensionales, pero es

necesario definir la componente vertical de la posición de la cámara, se igualará a la suma de la posición de la pelota y la distancia que había inicialmente entre ellas.

Dado que la distancia entre la cámara y la pelota se mide entre puntos tridimensionales, pero en este caso, por simplicidad, se usa directamente en el plano horizontal, la distancia entre la posición calculada de la cámara y la pelota será algo mayor de lo que sería en realidad. Sin embargo esta diferencia es pequeña, y para mantener el código más simple se ha considerado despreciable. Para predecir con exactitud total el resultado del giro habría que tener en cuenta tanto el ángulo formado entre el vector cámara-pelota y el plano horizontal como los radios de detección de los sensores "Near" de la cámara.

La comprobación de la presencia de obstáculos se realiza de la misma forma que en el tratamiento de los choques contra las paredes: mediante el método "rayCastTo" (ver apartado 5.5.2, página 64). En este caso el rayo se lanza desde la pelota, y en vez de apuntarlo hacia un objeto se hace hacia la posición que ocuparía la cámara, pasando dicha posición como parámetro. Esta comprobación se realizará una vez se dé la primera condición de activación del giro, es decir, solo cuando sea necesario. Si el método devuelve algún objeto no se harán más operaciones, ya que ello significa que habrá algún obstáculo o, si el objeto devuelto es la cámara, que ya está correctamente orientada.

En el Fragmento de código 28 aparece el código correspondiente; para probarlo en el fichero de ejemplo se debe, por una parte, activar el bloque de código "Giro automático en el plano horizontal – predecir posición", y por otra desactivar el usado en el apartado anterior.

```
# Asignaciones del nivel superior...

def main():
    # Resto de lógica...
    if teclado.events[events.IKEY] == 2:
        posicion_pelota =
Vector([pelota.worldPosition[0],pelota.worldPosition[1]])
        posicion_camara_hipotetica = posicion_pelota +
distancia_camara_pelota_inicial * -pelota['direccion_pelota']
        posicion_camara_hipotetica =
[posicion_camara_hipotetica[0],posicion_camara_hipotetica[1],altura_ca
mara_inicial]

        if not pelota.rayCastTo(posicion_camara_hipotetica):
            angulo_direccion_pelota =
obtener_angulo_x(pelota['direccion_pelota'])
            diferencia_angulos = obtener_angulo_diferencia(
angulo_camara_actual,angulo_direccion_pelota)
            velocidad_lateral = -diferencia_angulos *
FACTOR_GIRO_ORIENTAR_CAMARA
```

Fragmento de código 28. Comprobación de la presencia de obstáculos entre la posición que ocuparía la cámara si se hiciera el giro y la pelota.

5.7.3. Activación

Este sistema deberá seguir una determinada lógica de activación, que será distinta en función del tipo de juego o la utilidad que se le dé. En cada caso se tendrán en cuenta los estados distintas variables, presentes ya en el módulo o añadidas expresamente para esta finalidad, y será necesario hacer las modificaciones correspondientes en el código. A continuación se exponen algunos ejemplos útiles, y sus correspondientes bloques de código se encuentran en el módulo "control_camara_5.py", desactivados en primer lugar. Para probar cada uno, dado que son excluyentes, habrá que activarlo y desactivar el resto.

- Al pulsar una tecla: probablemente lo más simple, requerir que el usuario indique expresamente cuando desea orientar la cámara. Una implementación como la del apartado anterior sería suficiente, si bien en el Fragmento de código 29 se han modificado dos aspectos que podrían resultar de interés:
    - Se ha eliminado la necesidad de mantener la tecla pulsada para realizar el giro completo, añadiendo una variable global "orientando_camara", que toma el valor 1 al pulsar la tecla y se pone a 0 cuando el giro se ha completado (la diferencia de ángulos es menor que cierta cantidad).
    - Se ha añadido un término independiente a la ecuación que define la velocidad lateral, de manera que ésta no depende solamente del ángulo entre las direcciones, y se consigue un control más fino de su variación (se recomienda cambiar su valor para apreciar el efecto). Este término independiente debe ir acorde con el signo de la velocidad lateral, que a su vez es el contrario al de la diferencia de ángulos. En lugar de un término independiente podrían usarse otras opciones más complejas, como por ejemplo que el término añadido fuera dependiente de la velocidad de la pelota o de cualquier otra variable.

```
# Resto de asignaciones...

orientando_camara = 0

def main():
    global orientando_camara

    # Resto de lógica...

    if teclado.events[events.IKEY] == 1 or orientando_camara == 1:
        orientando_camara = 1
        posicion_pelota =
Vector([pelota.worldPosition[0],pelota.worldPosition[1]])
        posicion_camara_hipotetica = posicion_pelota +
distancia_camara_pelota_inicial * -pelota['direccion_pelota']
        posicion_camara_hipotetica =
[posicion_camara_hipotetica[0],posicion_camara_hipotetica[1],altura_ca
mara_inicial]
        if not pelota.rayCastTo(posicion_camara_hipotetica):
            angulo_direccion_pelota =
obtener_angulo_x(pelota['direccion_pelota'])
            diferencia_angulos =
obtener_angulo_diferencia(angulo_camara_actual,angulo_direccion_pelota
)
            termino_independiente = 5 if diferencia_angulos < 0 else -
5
            velocidad_lateral = -diferencia_angulos *
FACTOR_GIRO_ORIENTAR_CAMARA + termino_independiente
            if fabs(diferencia_angulos) < 0.1:
                orientando_camara = 0
```

Fragmento de código 29. Orientación de la cámara al pulsar una tecla.

- Continuamente: haciendo que el bloque de código se ejecute continuamente (excepto cuando el usuario mueve el ratón, dado que de otra manera el movimiento del ratón no tendría efecto) se consigue un sistema que complementa al seguimiento, y que no será percibido por el usuario como algo adicional al mismo. Hay que tener en cuenta tres aspectos:
    - No es viable usar un término independiente en la definición de la velocidad, como se hizo anteriormente. Si se usara un término independiente de valor 5, por ejemplo, la velocidad calculada pasaría de valer 5 a -5 alternativamente cuando la diferencia de ángulos fuera pequeña o nula. Si este salto se produjera un cada cuadro no sería perceptible, pero el suavizado lo retrasa unos pocos cuadros, lo que provoca un efecto de vaivén indeseable.
    - Cuando la pelota se desplaza lentamente los cambios de dirección son generalmente más grandes y frecuentes, lo que puede resultar molesto ya que la rapidez de los movimientos de la cámara depende de la magnitud de estos cambios. Como no se cuenta con el término independiente para afinar la relación entre ambos, una posible solución es modificar la constante "FACTOR_GIRO_ORIENTAR_CAMARA" en función de la velocidad lineal de la pelota, como se ha hecho en el Fragmento de código 30.

- El hecho de ejecutar continuamente esta lógica presenta un inconveniente cuando la pelota está parada: la cámara se mueve, en cuanto se detiene el ratón, para orientarse en la última dirección válida de la pelota, lo que imposibilita mantener un punto de vista diferente, por ejemplo para ver algo que sucede en la escena. Este problema se minimiza al redefinir la constante que controla el giro en función de la velocidad de la pelota (punto anterior), pero para atajarlo completamente lo más simple es, cuando la pelota avanza muy despacio, sustituir su propiedad "direccion_pelota" por la dirección cámara-pelota en cada cuadro (ver últimas líneas del Fragmento de código 30).

```
# Asignaciones del nivel superior...

def main():
    # Resto de lógica...

    if cuadros_sin_raton > 100:
        posicion_pelota =
Vector([pelota.worldPosition[0],pelota.worldPosition[1]])
        posicion_camara_hipotetica = posicion_pelota +
distancia_camara_pelota_inicial * -pelota['direccion_pelota']
        posicion_camara_hipotetica =
[posicion_camara_hipotetica[0],posicion_camara_hipotetica[1],altura_ca
mara_inicial]
        if not pelota.rayCastTo(posicion_camara_hipotetica):
            angulo_direccion_pelota =
obtener_angulo_x(pelota['direccion_pelota'])
            diferencia_angulos =
obtener_angulo_diferencia(angulo_camara_actual,angulo_direccion_pelota
)
            FACTOR_FINAL = FACTOR_GIRO_ORIENTAR_CAMARA *
pelota.linearVelocity.length / 20
            velocidad_lateral = -diferencia_angulos * FACTOR_FINAL

    if pelota.linearVelocity.length < 0.5:
        pelota['direccion_pelota'] = direccion_camara_actual
```

Fragmento de código 30. Orientación de la cámara continuamente.

- Si la diferencia entre las direcciones de la cámara y la pelota es mayor que un ángulo dado, y tras una determinada espera: usado por ejemplo en juegos de coches al dar marcha atrás, donde pasados unos instantes la cámara se gira para visualizar hacia dónde se avanza, o en juegos de acción donde se permite mirar atrás moviendo el ratón o *joystick* y después de un intervalo de tiempo sin hacerlo se recoloca la cámara detrás del personaje. En este caso la aplicación del giro depende, además de la posibilidad de llevarlo a cabo, como en apartados anteriores, de una variable global ("cuadros_sin_orientar") que representa los cuadros que han transcurrido (el tiempo de espera) desde que se debería haber realizado el giro, por ser el ángulo entre las direcciones mayor que el umbral, hasta el momento actual. Toda la lógica adicional está destinada a controlar esta variable, que define el comportamiento de la cámara; en este caso se ha decidido que el giro debe comenzar 100 cuadros después de que el ángulo entre ambas direcciones sea mayor que pi/2, e interrumpirse al mover el ratón. Asimismo, si en un momento dado el ángulo es mayor que el umbral pero antes de

esperar el tiempo definido (“cuadros_sin_orientar”) deja de serlo, se debe reiniciar este contador para la siguiente vez que se cumpla la condición esperar el tiempo adecuado de nuevo. Al igual que en el caso anterior, se debe corregir la propiedad que almacena la dirección de la pelota cuando ésta esté parada o se desplace muy despacio. En el Fragmento de código 31 aparece el código correspondiente:

```
# Resto de asignaciones...
cuadros_sin_orientar = 0

def main():
    global cuadros_sin_orientar
    # Resto de lógica...

    angulo_direccion_pelota =
obtener_angulo_x(pelota['direccion_pelota'])
    diferencia_angulos =
obtener_angulo_diferencia(angulo_camara_actual,angulo_direccion_pelota
)
    if fabs(diferencia_angulos) > pi/2:
        cuadros_sin_orientar += 1
    elif fabs(diferencia_angulos) < 0.2 or cuadros_sin_orientar < 100:
        cuadros_sin_orientar = 0
    if movimiento_raton:
        cuadros_sin_orientar = 0
    if cuadros_sin_orientar > 100:
        posicion_pelota =
Vector([pelota.worldPosition[0],pelota.worldPosition[1]])
        posicion_camara_hipotetica = posicion_pelota +
distancia_camara_pelota_inicial * -pelota['direccion_pelota']
        posicion_camara_hipotetica =
[posicion_camara_hipotetica[0],posicion_camara_hipotetica[1],altura_ca
mara_inicial]
        if not pelota.rayCastTo(posicion_camara_hipotetica):
            velocidad_lateral = -diferencia_angulos *
FACTOR_GIRO_ORIENTAR_CAMARA

    if pelota.linearVelocity.length < 0.5:
        pelota['direccion_pelota'] = direccion_camara_actual
```

Fragmento de código 31. Orientación de la cámara si la diferencia entre las direcciones de cámara y pelota es mayor que un ángulo dado y tras una espera.

5.8. Movimiento básico en vertical usando el ratón

Una vez se tiene un sistema como el explicado en los apartados anteriores, que, siempre en el plano horizontal, efectúa un seguimiento del objeto protagonista evitando obstáculos y adaptándose a su dirección, y permite al usuario cambiar el punto de vista, el siguiente paso es posibilitar los movimientos de la cámara en vertical. Con ello se pretende obtener un sistema que posibilite colocar el punto de vista (la cámara) en cualquier punto de una esfera de centro el objeto protagonista y radio la distancia inicial entre éste y la cámara.

En este apartado se explica cómo modificar el módulo diseñado anteriormente para que la cámara responda también a los movimientos del ratón en su eje Y, y más adelante se añadirán otros detalles. La complejidad del módulo crece significativamente, por lo que es conveniente entender el funcionamiento y la finalidad de cada bloque de código ya presente. Lo tratado en

este apartado se reflejará en el módulo "control_camara_6.py" del fichero de ejemplo ("demo_control_camara.blend"), cuya utilización habrá que configurar en el controlador "Python" correspondiente.

Dado que muchas de las funcionalidades relacionadas con el movimiento vertical que se implementen tendrán su equivalente en el lateral, es necesario renombrar la mayoría de las variables para reflejar a qué área pertenecen. Por ejemplo, la variable existente "margen_error_raton" pasará a llamarse "margen_error_raton_horizontal", y aparecerá una nueva llamada "margen_error_raton_vertical". Estos cambios ya se han realizado en el módulo "control_camara_6.py".

Aunque el paralelismo entre los sistemas que controlan los movimientos laterales y verticales es grande, hay dos diferencias fundamentales que hay que tener en cuenta:

- Debido a las características físicas de la cámara, se le debe aplicar velocidad vertical en todo momento (ver apartado 5.3.1, página 51), al contrario que sucede con la velocidad lateral. Esto significa que ya existe un sistema "automático", y hay que implementar correctamente desde un principio la transición entre éste y el manual (en el caso de la velocidad lateral no es necesario si no se pretende que la cámara se mueva por sí misma, por ejemplo al evitar obstáculos).
- Mientras que, sin considerar los obstáculos, la velocidad lateral siempre puede tomar cualquier valor, es decir, la cámara puede girar infinitamente alrededor del protagonista y puede colocarse en cualquier punto de la circunferencia que lo rodea, no pasa lo mismo con la vertical. Esta última no puede hacer que la cámara "suba" continuamente, ya que en ese caso se colocaría justo encima de la pelota, lo cual daría lugar a comportamientos erróneos (las coordenadas horizontales del vector cámara-pelota serían nulas, y fallarían todos los cálculos basados en ellas); lo mismo ocurre al desplazarse hacia abajo, cuando se encontraría con el suelo o en su defecto ocurriría la situación inversa a la anterior.

5.8.1. Aspectos comunes a la velocidad lateral

A continuación se describen, por orden, las partes del módulo que conciernen al movimiento en vertical y tienen un equivalente directo, ya explicado, en lo relativo al movimiento lateral. Asimismo se mencionan los cambios menores que se han hecho en el resto del *script*. El código correspondiente aparece en el Fragmento de código 32 y en el módulo "control_camara_6.py". Por comodidad, y dado que la complejidad aumenta notablemente, se proporciona activado por defecto; los bloques comentados son aquellos que se tratarán en sub-apartados posteriores. Este código es por otra parte suficiente para conseguir una respuesta correcta del movimiento de la cámara al del ratón.

- La adquisición de datos de la posición del ratón (primer bloque de la función "main") se ha separado del cálculo de la velocidad horizontal, de manera que sólo consiste en la obtención de la variable "movimiento_raton" y la recolocación del puntero.

- A continuación se calcula la magnitud de la velocidad vertical (“velocidad_vertical”) correspondiente al movimiento del ratón, siendo la lógica completamente equivalente a la usada al calcular la velocidad lateral (ver apartado 5.4.1, página 57). Se ha modificado la corrección del movimiento en el primer cuadro, que ha pasado a integrarse en la misma sentencia que la comprobación de que el movimiento vertical del ratón es mayor que el margen de error (“margen_error_raton_vertical”, cuyo cálculo y uso se cubren en el apartado 5.6.1, página 70). En este mismo bloque se actualiza la variable “cuadros_sin_raton_vertical”.
- En el siguiente bloque se efectúa el cambio entre los sistemas manual y automático. Si han transcurrido más de cierto número de cuadros sin mover el ratón verticalmente, se sustituye el valor presente en “velocidad_vertical”, que será nulo, por el necesario para que la cámara se coloque a la altura por defecto (la inicial). La explicación del cálculo de esta velocidad aparece en el apartado 5.3.1 (página 51). La constante que multiplica a la diferencia de alturas (que en este caso vale 1) interviene en un conflicto que se tratará en apartados posteriores.
- A continuación se limita la velocidad máxima, usando la constante “VELOCIDAD_VERTICAL_MAXIMA”, análogamente a como se explicó en el apartado 5.4.1 (página 57). En este caso, en vez de tratar por separado las velocidades positivas y negativas, se usa la función “copysign” de “math”, que devuelve el mismo valor que el pasado como primer parámetro con el signo del segundo parámetro.
- Posteriormente se realiza el suavizado de las paradas bruscas, explicado en la segunda parte del apartado 5.4.2 (página 60), y en este caso teniendo en cuenta la variable global “velocidad_vertical_anterior” (que se actualiza al final del módulo) y la constante “FACTOR_MANTENER_VELOCIDAD_VERTICAL”.
- Por último aparece el suavizado general, en el que se calcula la velocidad final a aplicar (“velocidad_vertical_final”) a partir de la media de cierto número de posiciones anteriores almacenadas en la cola “velocidad_vertical_deseada”. La explicación aparece en el apartado 5.4.2 (página 60).

```
from math import copysign
# Resto de asignaciones...

margen_error_raton_vertical = 1 / render.getWindowHeight()
cuadros_sin_raton_vertical = 0
velocidad_vertical_deseada = deque(maxlen = 10)
velocidad_vertical_anterior = 0

SENSIBILIDAD_RATON_VERTICAL = 500
FACTOR_MANTENER_RATON_VERTICAL = 0.7
MAXIMA_VELOCIDAD_VERTICAL = 70

def main():
    global cuadros_sin_raton_vertical
    global velocidad_vertical_anterior
    # Resto de logica...

    movimiento_raton = Vector([raton.position[0] -
0.5,raton.position[1] - 0.5])
    raton.position  = (0.5,0.5)

    # Velocidad vertical debida al movimiento del raton
    if fabs(movimiento_raton[1]) > margen_error_raton_vertical and
len(velocidad_vertical_deseada) > 0:
        velocidad_vertical = movimiento_raton[1] *
SENSIBILIDAD_RATON_VERTICAL
        cuadros_sin_raton_vertical = 0
    else:
        velocidad_vertical = 0
        cuadros_sin_raton_vertical += 1

    # Velocidad vertical para mantener la posición
    if cuadros_sin_raton_vertical > 100:
        altura_camara_actual = camara_principal.position[2]
        altura_camara_deseada = pelota.position[2] +
altura_camara_pelota_inicial
        velocidad_vertical = 1 * (altura_camara_deseada -
altura_camara_actual)

    # Limitar la velocidad vertical máxima
    if fabs (velocidad_vertical) > MAXIMA_VELOCIDAD_VERTICAL:
        velocidad_vertical =
copysign(MAXIMA_VELOCIDAD_VERTICAL,velocidad_vertical)

    # Evitar que la velocidad vertical se pare "en seco"
    if coinciden_signos(velocidad_vertical_anterior,
velocidad_vertical):
        velocidad_vertical = FACTOR_MANTENER_VELOCIDAD_VERTICAL *
velocidad_vertical_anterior

    # Suavizar la velocidad vertical
    velocidad_vertical_deseada.append(velocidad_vertical)
    velocidad_vertical_final =
calcular_media(velocidad_vertical_deseada)

    camara_principal.setLinearVelocity([velocidad_lateral_final,
velocidad_vertical_final,velocidad_lineal],1)

    velocidad_vertical_anterior = velocidad_vertical_final
```

Fragmento de código 32. Partes comunes entre el control de la velocidad vertical y la lateral.

5.8.2. Restricción del ángulo de elevación

Al lanzar el juego con el módulo de control de la cámara configurado según el apartado anterior, uno de los problemas más notorios es que al elevar la cámara se llega a un punto donde ésta comienza a girar descontroladamente.

La causa es que las coordenadas X e Y del vector cámara-pelota son prácticamente nulas, y cualquier movimiento causa grandes cambios en ellas. En cada cuadro el eje X de la cámara se alinea con un vector perpendicular al anterior y paralelo al plano horizontal, y consecuentemente realizará grandes giros acordes con los cambios en el vector. Estos giros ocasionan que las coordenadas horizontales del vector cámara-pelota cambien, entrando el sistema en un círculo vicioso.

La solución pasa por limitar el ángulo que el vector cámara-pelota forma con la vertical, deteniendo la velocidad vertical cuando dicho ángulo sea mayor que el umbral establecido. El vector se calcula mediante el método "getVectTo" (ver apartado7.3, página 136) y el ángulo usando el método "angle" del objeto de clase "Vector" obtenido anteriormente (ver apartado 7.2, página 134), pasando como parámetro el vector correspondiente al eje Z ([0,0,1]). Teniendo en cuenta los sentidos de ambos vectores, el ángulo resultante será de pi/2 radianes cuando la cámara esté a la misma altura que la pelota y de pi o 0 radianes cuando alcance las alturas máxima y mínima respectivamente. Los ángulos se convierten a grados mediante el método "degrees" del módulo "math" para facilitar el manejo.

Los casos a tener en cuenta son aquellos en los que el ángulo formado es cercano a pi o a 0 radianes, en los cuales se parará la velocidad vertical en sentido positivo o negativo respectivamente. Una opción sería evitar las velocidades verticales negativas para ángulos cercanos a pi/2 rad, para que la cámara no atravesase el suelo, pero entonces no se comportaría correctamente en situaciones donde sí se deben poder adquirir ángulos menores, por ejemplo cuando la pelota realiza grandes saltos. Por tanto se ha decidido considerar sólo los casos de pi y 0 radianes, y tratar el suelo como un obstáculo cualquiera, lo cual se explicará más adelante.

Aunque se podría restringir directamente la magnitud "velocidad_vertical" una vez calculada, se aprovecha el mismo sistema que el dedicado al control de los choques, similar al usado en el movimiento lateral y que se implementará más adelante. Es decir, cuando el ángulo supere el umbral establecido se cambiará la variable "parar_velocidad_vertical" en lugar de anular directamente "velocidad_vertical".

En el Fragmento de código 33 se incluyen los dos bloques de código a añadir para limitar el ángulo formado con la vertical. El primero podría colocarse en cualquier punto (siempre antes que el segundo), mientras que el segundo debe situarse después del cálculo de la variable "velocidad_vertical_final". En el módulo "control_camara_6.py" se deben activar ambos, eliminando las comillas triples que los encierran; el primero se ha llamado "Limitar el ángulo formado con el eje Z (básico)", y el segundo "Parar la velocidad vertical (básico)".

```
from math import degrees
# Resto de asignaciones...

def main():
    #Resto de lógica...

    angulo_vertical =
degrees(camara_principal.getVectTo(pelota.position)[1].angle([0,0,1]))
    if angulo_vertical > 170:
        parar_velocidad_vertical = 1
    else:
        parar_velocidad_vertical = 0

    if coinciden_signos(velocidad_vertical_final,
parar_velocidad_vertical):
        velocidad_vertical_final = 0
```

Fragmento de código 33. Limitación del ángulo de elevación de la cámara.

Es posible hacer una pequeña modificación para suavizar la detención de la cámara cuando se alcanza el ángulo límite, permitiendo que la variable "parar_velocidad_vertical" tome cualquier valor entre -1 y 1, en función del ángulo formado, y multiplicando la velocidad vertical a aplicar a la cámara por el inverso de dicho valor. De esta forma se mantiene la compatibilidad con los casos donde la variable de control solo tome los valores extremos, por ejemplo el control de choques.

Para decidir el valor de "parar_velocidad_vertical" se sigue, en el caso del ejemplo, la siguiente lógica: si el ángulo es menor de 160º se permite el movimiento libre, "parar_velocidad_vertical" vale 0. Si el ángulo supera 160º, la variable de control se define como la diferencia entre el ángulo formado y 160 dividida entre 15, lo que resulta en un valor positivo entre 0 (ángulo de 160º) y 1 (ángulo de 175º). Para ángulos menores de 20º se usa una estructura equivalente.

La modificación de la velocidad "velocidad_vertical_final" en función de esta variable se efectúa cuando sus signos son iguales, en cuyo caso se multiplica el valor de la velocidad a aplicar por la diferencia entre la unidad y el valor absoluto de "parar_velocidad_vertical".

En el Fragmento de código 34 aparece el sistema anterior modificado para incluir este suavizado. En el módulo "control_camara_6.py" del fichero de ejemplo se deben anular los bloques que se activaron en el apartado anterior y activar los que les sustituyen (es decir, "Limitar el ángulo formado con el eje Z (suavizado)" y "Parar la velocidad vertical").

```
# Resto de asignaciones...

def main():
    #Resto de lógica...

    angulo_vertical =
degrees(camara_principal.getVectTo(pelota.position)[1].angle([0,0,1]))
    if angulo_vertical > 160:
        parar_velocidad_vertical = (angulo_vertical - 160) / 15
    elif angulo_vertical < 20:
        parar_velocidad_vertical = (angulo_vertical - 20) / 15
    else:
        parar_velocidad_vertical = 0

    if coinciden_signos(velocidad_vertical_final,
parar_velocidad_vertical):
        velocidad_vertical_final = (1 - fabs(parar_velocidad_vertical)
) *  velocidad_vertical_final
```

Fragmento de código 34. Limitación del ángulo de elevación de la cámara con parada suavizada.

5.8.3. Adaptación de la velocidad lateral al ángulo de elevación

Al permitir el movimiento de la cámara en vertical se pone de manifiesto una consecuencia negativa de la forma en que se calcula la velocidad lateral: como la velocidad a aplicar se obtiene directamente del movimiento del ratón, movimientos iguales causarán velocidades iguales en todos los casos, independientemente de la elevación de la cámara. Como el radio de la circunferencia que recorre la cámara al girar alrededor de la pelota cambia según su altura relativa, cuando esté en una posición elevada respecto a la misma, la velocidad angular que produce cierto movimiento del ratón será mayor que la que ocasionada por el mismo movimiento cuando la cámara está a la misma altura que la pelota. Es decir, cuando la cámara se eleva gira más con el mismo movimiento del ratón, lo que se traduce en un cambio en su sensibilidad, que por lo general es indeseable.

La solución más simple pasa por aplicar a la velocidad lateral un factor de reducción basado en la elevación de la cámara. Este factor se calcula teniendo en cuenta el ángulo que forma el vector cámara-pelota con el eje Z (variable “angulo_vertical”, calculada en el apartado anterior), y debe tomar valores entre 0 y 1. Se multiplicará por la magnitud de la velocidad lateral, por lo que debe valer 1 cuando la elevación sea nula (cámara y pelota contenidas en un mismo plano horizontal), de manera que no reduzca la velocidad, y tender a 0 cuando dicha elevación sea grande. Como se comentó en el apartado anterior, una elevación nula corresponde a un ángulo “angulo_vertical” de 90º, y una elevación máxima a 180º (arriba) o 0º (abajo).

El ángulo absoluto de elevación se obtiene mediante el valor absoluto de la diferencia entre el valor de “angulo_vertical” y 90º. Para hacer que varíe entre 0 y 1 basta con dividirlo entre 90, y para hacer que el valor 1 corresponda con elevaciones nulas y viceversa se debe restar, de la unidad, el cociente obtenido. La relación entre el ángulo y el factor obtenido, que de esta manera es lineal, puede afinarse añadiendo modificaciones a la ecuación anterior. En

el ejemplo se ha elevado el cociente al cuadrado, de manera que el factor es prácticamente nulo mientras las elevaciones son moderadas. Se ha tenido en cuenta que la elevación sufre cambios constantemente al mover el ratón, incluso aunque sólo se pretenda hacer en horizontal, y se ha considerado que al usar una relación lineal estos cambios involuntarios ocasionan variaciones indeseables en la velocidad lateral.

En el Fragmento de código 35 aparece el código correspondiente a la corrección. El cálculo del factor puede colocarse en cualquier lugar, pero la corrección propiamente dicha debe hacerse en la sección de la velocidad lateral, concretamente después del bloque que evita que se pare en seco (si se hiciera antes, reducciones grandes debidas a cambios rápidos de altura mientras se gira no tendrían efecto) y antes del suavizado (si se hiciera después se anularía su función, apareciendo brusquedades). Para mantener el código organizado, el cálculo del factor se ha colocado junto a la corrección de la velocidad. En el módulo "control_camara_6.py" se deben quitar las comillas triples de dicho bloque de código, denominado "Adaptar la velocidad lateral a la elevación de la cámara".

```
# Asignaciones del nivel superior...

def main():
    #Resto de lógica...

    # Evitar que la velocidad lateral se pare "en seco"...

    elevacion_camara = 1 - (fabs(angulo_vertical - 90) / 90)**2
    velocidad_lateral = elevacion_camara * velocidad_lateral

    # Suavizar la velocidad lateral...
```

Fragmento de código 35. Corrección de la velocidad lateral en función del ángulo de elevación de la cámara.

5.8.4. Uso de coordenadas globales

Al posibilitar el control vertical de la cámara mediante el movimiento del ratón se pone de manifiesto una nueva consecuencia negativa de aplicar la velocidad usando los ejes globales como referencia, que se une a la comentada en el apartado 5.5.4 (página 67).

Como se comentó anteriormente, debido a que ni el eje Y de la cámara es completamente vertical, ni el Z es completamente paralelo al plano horizontal, al aplicar velocidad vertical hacia arriba también se hace hacia adelante ligeramente y cuando se hace hacia adelante también se impulsa hacia abajo.

El problema que aparece en este caso se deriva de la decisión de compromiso que hubo que adoptar al establecer la constante en la ecuación que calcula la velocidad vertical necesaria para mantener la altura original: `velocidad_vertical = CONSTANTE * (altura_camara_deseada - altura_camara_actual)`. Valores pequeños de esta constante permiten que cuando el protagonista se desplaza muy rápido la velocidad vertical no sea capaz de compensar el empuje hacia abajo causado por la componente vertical de la velocidad lineal. En apartados anteriores fue suficiente con aumentar su valor hasta 10 para contrarrestar

este efecto, sin embargo al habilitar el control en vertical no es posible hacerlo por los siguientes motivos:

- Si, cuando el sistema automático entra a funcionar, después de no mover el ratón durante 100 cuadros, la cámara está muy elevada, la diferencia entre alturas que determina la velocidad vertical es grande. Si se multiplica por una constante también grande, el movimiento generado es muy brusco.
- Dado que en este caso la velocidad vertical se somete a un suavizado antes de ser aplicada, siempre existe cierto retraso en la respuesta. Este retraso hace que en el momento en que la velocidad aplicada debiera ser nula, por haberse alcanzado la altura objetivo, se siga desplazando. Habrá por tanto una respuesta a este desplazamiento erróneo, que de nuevo se retrasará, volviendo a comenzar el ciclo. La magnitud de esta constante determina si la oscilación, que es inherente al método usado en el suavizado, es perceptible o no. Valores mayores la hacen más notoria, e incluso llevado al extremo la oscilación puede hacerse inestable.

La solución tanto a este problema como al comentado en el apartado 5.5.4 (página 67) (no es posible parar la velocidad lineal cuando es necesario ya que en ella influye la velocidad vertical) es aplicar la velocidad sin apoyarse en los ejes de la cámara.

La base del proceso es similar a la usada en el control del objeto protagonista: se calcula en cada cuadro la dirección global en que cada velocidad para que su efecto sea el deseado. Posteriormente cada vector de dirección se multiplica por la constante correspondiente a cada velocidad, se suman y el resultado se pasa al método "setLinearVelocity".

Las constantes a utilizar son las que anteriormente se colocaban en cada componente del vector que se pasaba a "setLinearVelocity", por tanto no es necesario ningún cambio. Las direcciones se calculan tomando como base la orientación de los ejes de la cámara, que se obtiene de su matriz "worldOrientation" (ver apartado 7.8, página 147), eliminando las dependencias entre ellos según sea necesario en cada caso. Para mejorar la legibilidad se extrae primeramente la dirección de cada eje de dicha matriz, almacenándolas en las variables "eje_x_camara", "eje_y_camara" y "eje_z_camara". A continuación se detalla el proceso:

- La dirección correspondiente a la velocidad lateral será siempre un vector equivalente al eje X de la cámara, es decir, perpendicular la línea que une la cámara y la pelota, hacia la derecha, y paralelo al plano horizontal.
- La dirección de la velocidad lineal ("direccion_velocidad_lineal") será, por lo general, un vector con la misma dirección en el plano horizontal que el eje Z de la cámara (y su sentido negativo) pero paralelo a dicho plano, es decir, un vector equivalente al eje mencionado con la tercera componente anulada. De esta manera la velocidad lineal no influye en la vertical. Sin embargo, en los casos donde la inclinación de la cámara sea grande, la dirección de la velocidad lineal no puede ser paralela al plano horizontal, ya que su función es controlar la distancia entre la cámara y la pelota. En estos casos sí es necesario que la dirección de la velocidad lineal sea equivalente a la del eje Z de la

cámara, para que realmente avance hacia ella. Un ángulo de elevación grande solamente se puede dar si la pelota se desplaza rápidamente en vertical o si el usuario mueve la cámara con el ratón; por tanto, la decisión de qué dirección escoger para la velocidad lineal dependerá de que la componente vertical de la velocidad de la pelota sea mayor o menor que un valor dado (el 0 no es práctico porque siempre se producen pequeñas variaciones) y del tiempo que el usuario lleva sin mover el ratón en vertical.

Por otra parte, la influencia que la velocidad lineal ejerce sobre la vertical sí podría considerarse beneficiosa cuando la pelota avanza hacia atrás, en cuyo caso provoca que la cámara se eleve ligeramente antes o mientras gira para adoptar la nueva dirección. Para habilitar esta distinción es suficiente con igualar "direccion_velocidad_lineal" a "-eje_z_camara" cuando la magnitud de la velocidad lineal calculada sea negativa, es decir la cámara avance hacia atrás. Sin embargo esto implica una pérdida de control sobre la velocidad vertical, que en apartados posteriores, donde se tienen en cuenta los choques contra obstáculos en todas las direcciones, resulta perjudicial. Por tanto, aunque en el ejemplo sí se incluye, en los siguientes apartados no se hará. En la práctica podría usarse en escenarios sin obstáculos elevados, o donde no se tengan en cuenta, o modificar la lógica para considerar la presencia o cercanía de obstáculos además del sentido de la velocidad lineal.

- La dirección de la velocidad vertical debe ser, en general, equivalente a la del eje Y de la cámara, de manera que siempre la impulse correctamente. Si no se posibilitara el movimiento en vertical mediante el ratón la velocidad para mantener la posición podría aplicarse sólo en el eje Z, de manera que no influyera en la lineal. Sin embargo, dado que la cámara se mueve en una esfera alrededor del protagonista, la velocidad vertical (al igual que la horizontal) debe ser tangente a la misma, es decir, equivalente al eje Y. Al usar esta dirección para la velocidad vertical se sigue interfiriendo en la lineal, y aparece igualmente el problema de no poder detenerla cuando es necesario. Para solucionarlo puede por ejemplo determinarse que cuando se dé la condición que hace parar la velocidad lineal (hay un obstáculo detrás de la cámara) la dirección de la velocidad vertical sea paralela al eje Z.

En el Fragmento de código 36 aparece todo el código necesario para hacer el cambio a coordenadas globales, incluyendo las consideraciones anteriores. El cálculo de los vectores finales debe hacerse después de haber obtenido todas las magnitudes, mientras que el de las direcciones podría en un principio hacerse en cualquier punto del módulo (antes que el cálculo de los vectores). Sin embargo éste se hace al principio, ya que las direcciones se usarán en la parte del control de choques, que se cubre en el apartado siguiente.

Este código aparece en el fichero y módulo de ejemplo ("control_camara_6.py") entre comillas triples. Para poder usarlo es necesario activar cada uno de los bloques: el cálculo de direcciones en la parte superior (llamado "Calcular las direcciones globales correspondientes a las locales"), el ajuste de las mismas según la velocidad lineal (denominado "Correcciones a

las direcciones calculadas") en la parte central, y al final el cálculo de los vectores correspondientes a cada velocidad ("Calcular vectores velocidad"), y la aplicación de la velocidad respecto a coordenadas globales ("Aplicar velocidad total"). Además, se debe anular la línea que anteriormente aplicaba la velocidad respecto a coordenadas locales.

```
# Asignaciones del nivel superior...

def main():
    orientacion_camara = camara_principal.worldOrientation.copy()
    orientacion_camara.transpose()
    eje_x_camara,eje_y_camara,eje_z_camara = orientacion_camara[:]

    if cuadros_sin_raton_vertical < 100 or
fabs(pelota.worldLinearVelocity[2]) > 1:
        direccion_velocidad_lineal = -eje_z_camara
    else:
        direccion_velocidad_lineal = Vector((-eje_z_camara[0],-
eje_z_camara[1],0))
        direccion_velocidad_lineal.normalize()
    direccion_velocidad_lateral = eje_x_camara
    direccion_velocidad_vertical = eje_y_camara

    #Resto de lógica...

    if pared_cerca and velocidad_lineal > 0:
        velocidad_lineal = 0
        # Hacer velocidad lineal independiente de vertical
        direccion_velocidad_vertical = Vector([0,0,1])
    if velocidad_lineal > 0:
        # Elevar cámara al avanzar hacia atrás
        direccion_velocidad_lineal = -eje_z_camara

    #Resto de lógica...

    vector_velocidad_lineal = -velocidad_lineal *
direccion_velocidad_lineal
    vector_velocidad_vertical = velocidad_vertical_final *
direccion_velocidad_vertical
    vector_velocidad_lateral = velocidad_lateral_final *
direccion_velocidad_lateral
    vector_velocidad_total = vector_velocidad_lineal +
vector_velocidad_lateral + vector_velocidad_vertical

    camara_principal.setLinearVelocity(vector_velocidad_total,0)
```

Fragmento de código 36. Aplicación de la velocidad usando coordenadas globales como referencia, gestión de las dependencias entre ejes.

5.9. Control de choques en cualquier dirección

En los apartados anteriores se ha explicado cómo tratar los choques que se producen cuando la cámara se mueve solamente en el plano horizontal, mediante dos sensores: uno dedicado a la velocidad lateral, y otro a la lineal. Al posibilitar el control manual en vertical se hace necesario introducir un mecanismo que evite que la cámara atraviese las paredes también en esta dirección, y por combinación con las velocidades lateral y lineal en cualquiera del espacio tridimensional.

En este apartado se trata todo lo referente a dicho mecanismo, y en el módulo "control_camara_7.py" del fichero de ejemplo ("demo_control_camara.blend") se ha implementado el código correspondiente, de manera que se puede probar configurando el controlador "Python" dedicado al control de la cámara para que ejecute dicho módulo ("control_camara_7.main").

5.9.1. Introducción

Durante la realización del proyecto se han estudiado diversas aproximaciones a la resolución del problema, cuya complejidad es notablemente mayor que antes de incorporar el movimiento vertical. Entre todas ellas hay una que se ha considerado la más simple, práctica y completa, que se tratará en profundidad en este apartado. Además se comentará anteriormente otra que, aunque finalmente se ha descartado por fallar en determinados aspectos, podría ser adecuada para la mayor parte de los juegos, y probablemente sea más eficiente.

Cabe comentar la razón por la que se ha descartado el razonamiento que a priori podría parecer más lógico: una vez se ha detectado la proximidad de una pared, no permitir que se apliquen a la cámara velocidades verticales ni laterales del mismo signo que las que se aplicaron en el cuadro anterior al choque. Aunque es el método óptimo en los choques contra paredes oblicuas, tales como rampas, falla con los obstáculos más comunes, que son paredes verticales o superficies horizontales (por ejemplo el suelo): en estos casos hace que se restrinja el movimiento en una dirección en la que no debería hacerse; por ejemplo, si la cámara choca contra una pared vertical, solamente se debe evitar la velocidad lateral que haga que se acerque más a dicha pared, mientras que la vertical debe mantenerse activa.

Dedicar un sensor "Near", con diferentes radios de detección, a cada tipo de velocidad no representa ninguna ventaja respecto a lo anterior, ya que la raíz del problema es la imposibilidad de configurarlos para que sólo respondan a superficies horizontales o verticales.

El funcionamiento básico de los dos métodos que se tratan es el siguiente:

- Una vez detectado contacto, trazar mediante el método "rayCast" un rayo desde la cámara hasta el/los objetos que han interferido con la esfera de detección. La normal a la superficie del objeto en el punto en que el rayo la atraviesa (uno de los parámetros devueltos por el método) permite conocer la orientación de dicha superficie, y decidir qué velocidad/es parar en consecuencia.
- Cuando se detecte contacto, trazar un rayo en cada una de las direcciones básicas que podrán adoptar las velocidades vertical y lateral (arriba, abajo, derecha e izquierda; se puede incluir la lineal también), de longitud similar al radio de detección del sensor "Near". Si alguno de estos rayos detecta un objeto, no se debe permitir el movimiento en su dirección y sentido.

Dado que ambos hacen uso del método "rayCast", su funcionamiento básico se explica en primer lugar.

5.9.2. Uso del método “rayCast”

Este método, presente en todos los objetos del juego, consiste en la versión completa de “rayCastTo” (ver apartado 5.5.2, página 64). Su funcionamiento básico es equivalente, aunque mediante la aceptación y devolución de más cantidad de parámetros adopta una flexibilidad mucho mayor que permite hacer cálculos más complejos.

Los parámetros que admite aparecen a continuación:

- Objeto hacia cuyo origen se traza el rayo, o punto del espacio (vector tridimensional u objeto de tipo “tuple” o “list”). Equivalente al parámetro pasado anteriormente a “rayCastTo”.
- Objeto desde el que se lanza el rayo, o punto del espacio. Este parámetro hace al rayo independiente del objeto desde el que se llama al método, ya que tanto el punto donde empieza como donde termina pueden definirse libremente. Si se omite, o se pasa “None”, el rayo se lanza desde el objeto que llamó al método.
- Distancia hasta la cual se detectan intersecciones (número real). Permite limitar o extender la búsqueda: el objeto hacia el que se traza el rayo sólo se usa para calcular la dirección, mientras que la distancia indica si el rayo llega hasta el objeto o lo sobrepasa o, si es negativa, hace que se lance en sentido contrario. Si se omite, pasando 0, el rayo llega hasta el objeto pasado como primer parámetro.
- Propiedad buscada (cadena de caracteres). Hace que sólo se detecten los choques con objetos que tengan una propiedad cuyo nombre coincida con el pasado.
- Orientación de la normal (0 o 1). Si se pone a 1 la normal que se devuelve será la normal real de la cara donde se ha producido el choque. Si su valor es 0 o se omite, siempre se devuelve la normal que apunta hacia el lado de la superficie donde impacta el rayo. Por regla general las normales de las caras de los objetos bien construidos apuntan hacia el exterior, por lo que el valor que se pase no supondrá ninguna diferencia en el resultado. Sin embargo es conveniente que sea 0 u omitirlo, para evitar comportamientos erróneos en caso de que haya algún error en las normales.
- Atravesar objetos (0 o 1). Sólo tiene sentido cuando se buscan objetos que tengan una determinada propiedad. Si se pasa un 1 se descartan los objetos que no tengan la propiedad y se sigue buscando hasta llegar a la distancia límite o hasta encontrar uno, mientras que si se omite o se pasa un 0 el rayo se detiene en el primer choque.
- Datos devueltos (0, 1 o 2). Determina qué datos se incluyen en el objeto que devuelve el método (ver siguiente lista). El valor por omisión es 0.

De estos parámetros sólo el primero es obligatorio, mientras que el resto tomarán los valores por defecto si se omiten. Solamente es necesario especificar los parámetros, en orden, hasta el último de ellos cuyo valor no sea el predeterminado (pasando, en los anteriores a este último que no requieran modificaciones, el valor por omisión). Así, si por ejemplo se quisiesen detectar objetos con una determinada propiedad pero no definir una distancia límite

ni un punto de origen diferente, la llamada quedaría: `resultado = objeto_origen.rayCast(objeto_final, None, 0, 'propiedad_buscada')`.

El objeto que devuelve el método es siempre de tipo "tuple", y el número de elementos que contiene depende del último parámetro de la lista anterior. Según valga 0, 1 o 2 se incluirán, respectivamente, los tres primeros, los cuatro primeros o todos los elementos que aparecen a continuación:

- Objeto detectado: objeto del juego (tipo "KX_GameObject") contra el que ha chocado el rayo. Si no ha chocado, contiene "None".
- Punto de choque: vector tridimensional con las coordenadas globales del punto en el que el rayo toca la superficie del objeto. Si no ha chocado, contiene "None".
- Normal: vector normal a la superficie del objeto en el punto donde impacta el rayo. Si no ha chocado, contiene "None".
- Cara del objeto con la que ha chocado: objeto de tipo "KX_PolyProxy", que contiene los datos de dicha cara, tales como el material asociado o referencias a los datos de sus vértices. Si no ha chocado, contiene "None".
- Coordenadas en el mapa UV del punto de choque. Si no ha chocado, contiene "None".

Dado que la única funcionalidad extra que se requiere en este apartado respecto a "rayCastTo" es la obtención de la normal, las llamadas al método se harán omitiendo o poniendo a 0 el último parámetro, de forma que el objeto "tuple" conste de tres elementos.

Un aspecto importante a tener en cuenta es que, cuando el punto de inicio se encuentra dentro del objeto de destino, o éste tiene forma cóncava, es probable que el rayo no atraviese ninguna superficie, y por tanto el método no devuelva ningún objeto válido. En el escenario del proyecto este hecho tiene relevancia por ejemplo en las rampas que se han colocado, o en el recinto circular con una rampa en su interior.

5.9.3. Toma de decisiones basada en la orientación de las superficies detectadas

El razonamiento básico es el siguiente: cuando un sensor "Near" detecte proximidad, acceder a su atributo "hitObjectList" y lanzar un rayo desde la cámara a cada uno de los objetos detectados. Obtener el vector normal a la superficie en el punto de choque, calcular mediante "angle" el ángulo que forma con el plano horizontal (inclinación) y el que su proyección horizontal forma con la de la dirección de apunte de la cámara, y tomar las siguientes decisiones:

- Si el valor absoluto de la inclinación es mayor que 5º (u otro valor pequeño), parar la velocidad vertical en el sentido del desplazamiento vertical realizado en el cuadro anterior.

- Si el valor absoluto de la inclinación es menor que 85º (o similar), y el ángulo entre la proyección horizontal de la normal y la dirección de la cámara es mayor que 5º, parar la velocidad lateral en el sentido en que giraba la cámara anteriormente.

Es necesario dejar unos pequeños márgenes para tener en cuenta los posibles errores de redondeo, y despreciar las inclinaciones irrelevantes. Por ejemplo, si se parase la velocidad lateral cuando el ángulo fuese menor de 90º, y en un momento dado la normal de una superficie horizontal tiene una inclinación de 89.9º, se restringiría la velocidad lateral correspondiente, cuando en realidad esa superficie no representa un obstáculo para ella.

Antes de abordar la implementación del código hay que tener en cuenta los siguientes aspectos respecto a los apartados anteriores:

- Dado que se unifica el control de los choques laterales y verticales, se debe suprimir el bloque de código que en apartados anteriores gestionaba la variable "parar_velocidad_lateral".
- La sección que impide que la cámara se incline más de un determinado ángulo seguirá activa, pero se debe eliminar la última parte de la estructura "if/else", que pone la variable "parar_velocidad_vertical" a cero si no se dan las condiciones anteriores (ver Fragmento de código 33, página 86), de manera que no interfiera con las decisiones tomadas en el control de choques.
- Al incluir en la lógica que determina si se debe parar la velocidad lateral el ángulo formado entre las componentes horizontales de la normal y las del apunte de la cámara, el uso de un sensor "Near" diferente para gestionar la velocidad lineal pierde su razón de ser (ver apartado 5.5.4, página 67). Aunque los dos *logic bricks* asociados al objeto "camara_principal" no se cambiarán, para que los apartados anteriores sigan funcionando, en el módulo "control_camara_7.py" solamente se usará uno de ellos, y la referencia al mismo se denominará "sensor_pared_cerca".

En cuanto a la manera de actualizar las variables globales "parar_velocidad_lateral" y "parar_velocidad_vertical", se ha llegado a la conclusión de que lo óptimo es hacerlo solamente cuando la lista de objetos detectados por el sensor "Near" cambie. Para ello, en lugar de decidir si la lógica se ejecuta a partir del atributo "positive" del sensor, se hace según el resultado de comparar la lista de objetos detectados actual con la del cuadro anterior (almacenada en una variable global). Como el atributo "hitObjectList" es de tipo "CListValue", interno de Blender y sin método "copy", en primer lugar se convierte a un nuevo objeto tipo "list", que será el que se almacene y se use para comparar. Ver Fragmento de código 37 más adelante.

A continuación se detallan los pasos necesarios para añadir el control total de choques, cuyo código correspondiente aparece en el Fragmento de código 37:

- Calcular el sentido de giro de la cámara (explicado en el apartado 5.6.2, página 71). Si se está desplazando hacia la derecha (girando hacia la izquierda) la variable "sentido_giro_camara" 1, en caso contrario -1.

- Calcular el sentido desplazamiento vertical entre el cuadro anterior y el actual, usando la variable global "posicion_camara_anterior" que se actualiza en cada cuadro. Sólo interesa el signo, por tanto se usa la función "copysign", pasando como parámetros el número 1 y la diferencia entre las posiciones verticales actual y anterior. Si en vez del desplazamiento se usara la velocidad vertical aplicada en el cuadro anterior el resultado sería erróneo en los casos donde, debido a la gravedad, la velocidad se aplicó en un sentido pero la cámara se movió en el contrario.
- Generar una lista ("objetos_detectados") a partir del atributo "hitObjectList" del sensor "Near", y acto seguido compararla con la que se obtuvo en el cuadro anterior. Si son diferentes se entra en el bloque lógico para re-calcular las velocidades que se deben anular, si no éstas mantendrán sus valores actuales.
- Una vez se produzca un cambio en los objetos que hacen contacto, se deben reiniciar los valores de "parar_velocidad_lateral" y "parar_velocidad_vertical": por ejemplo, si el cambio consiste en que el objeto que requería restringir la velocidad vertical ya no es detectado, pero no se reiniciaran las variables, "parar_velocidad_vertical" seguiría valiendo 1 o -1.
- Posteriormente se entra en un bucle "for" que recorre la lista "objetos_detectados", donde en primer lugar se añade a cada objeto una propiedad (por ejemplo "detectado") y acto seguido se traza un rayo desde la cámara hasta su origen, indicando que el objeto contra el que choque debe tener dicha propiedad, para asegurar que la normal obtenida no corresponde a una superficie ajena al objeto que se haya interpuesto en el camino del rayo. Para ello se usa el método "rayCast", explicado en el apartado anterior, teniendo en cuenta que el parámetro que ordena atravesar objetos ("xray" en inglés) debe activarse. El vector normal a la superficie obtenido se almacena en una variable, "normal" en este caso.
- Antes de operar con este vector, se tiene en cuenta el caso del suelo, dado que la lógica que se usará para determinar la velocidad a restringir no es aplicable en su caso: independientemente de la presencia de obstáculos, y del movimiento de la cámara, la detección del suelo siempre deberá implicar que se eviten velocidades verticales negativas.
- Si el objeto no se trata del suelo, y la normal almacenada es válida (lo cual no ocurre cuando la cámara se encuentra "dentro" de los objetos, es decir, ninguna superficie se interpone entre ella y el origen), se procede a determinar las velocidades a evitar. Si no es válida (vale "None") las opciones son parar ambas velocidades o no hacerlo con ninguna, ya que no se dispone de datos para tomar la decisión correcta.
- Para determinar qué velocidades evitar es necesario en primer lugar calcular la inclinación de la normal ("inclinación_normal") mediante el ángulo formado con el eje Z. El resultado debe ser un ángulo entre 0 rad, si es paralela al plano horizontal (la superficie es vertical), y pi/2 rad, si es paralela a dicho eje (la superficie es horizontal), siendo indiferente la coincidencia o no de sentidos.

- Posteriormente se calcula el ángulo (“angulo_horizontal_normal_apunte”) que la dirección de apunte de la cámara (obtenida anteriormente, ver apartado 5.6.2, página 71) forma con las componentes horizontales de la normal. En los casos en que estas componentes sean exactamente nulas (superficies horizontales) el método “angle” devolverá error, por lo que la solución más cómoda es encerrar el cálculo en una estructura “try/except”. Cuando se produzca un error, se dará por hecho que el ángulo es 0, de manera que posteriormente se permita la velocidad lateral (adecuado si la superficie es horizontal).
- Por último se toman las decisiones correspondientes en función de la inclinación de la normal y el ángulo anterior:
    - Si la inclinación de la normal es mayor que un ángulo pequeño, la variable “parar_velocidad_vertical” se iguala al signo del desplazamiento vertical de la cámara, calculado anteriormente.
    - Si la inclinación es menor que un ángulo cercano a 90º, y el ángulo entre el apunte de la cámara y las componentes horizontales de la normal es mayor que un cierto margen, la variable “parar_velocidad_lateral” se iguala al signo de “sentido_giro_camara”, también calculado anteriormente.
- En cada bloque de los anteriores se comprueba la presencia de obstáculos. Si la pelota no es visible, la velocidad a detener es la contraria a la calculada.

En el Fragmento de código 37 aparece todo el código que se debería modificar respecto al de los apartados anteriores para implementar el control de choques mediante este método. En el módulo “control_camara_7.py” del fichero de ejemplo se ha incluido todo el código necesario, y el correspondiente a este método se ha activado por defecto, por tanto no hay que hacer ninguna modificación para probarlo.

```
# Resto de asignaciones...

posicion_camara_anterior = camara_principal.worldPosition.copy()
objetos_detectados_anteriores = []

sensor_pared_cerca = camara_principal.sensors['cerca_lateral']
sensor_pared_cerca.distance = 1.5
sensor_pared_cerca.resetDistance = 1.5

def main():
    global parar_velocidad_lateral
    global parar_velocidad_vertical
    global posicion_camara_anterior

    pared_cerca = sensor_pared_cerca.positive

    # Calcular el sentido de giro de la cámara...

    desplazamiento_vertical_camara = copysign(1,
camara_principal.position[2] - posicion_camara_anterior[2])

    objetos_detectados = list(sensor_pared_cerca.hitObjectList)
    if objetos_detectados != objetos_detectados_anteriores:
        parar_velocidad_lateral,parar_velocidad_vertical = 0,0
        for objeto in objetos_detectados:
            objeto['detectado'] = 1
            normal = camara_principal.rayCast(objeto,None,0,
'detectado',0,1)[2]
            if objeto.name == 'suelo':
                parar_velocidad_vertical = -1
            elif normal:
                inclinacion_normal = fabs(normal.angle([0,0,1]) -
pi/2)
                normal_proyectada_horizontal = Vector(normal[0:2])
                angulo_direccion_camara_proyeccion_normal =
normal_proyectada_horizontal.angle(direccion_camara_actual)
                if inclinacion_normal > radians(5):
                    parar_velocidad_vertical =
desplazamiento_vertical_camara
                    if hay_obstaculo:
                        parar_velocidad_vertical = 0 -
parar_velocidad_vertical
                if inclinacion_normal < radians(85):
                    parar_velocidad_lateral = sentido_giro_camara
                    if hay_obstaculo:
                        parar_velocidad_lateral = 0 -
parar_velocidad_lateral

    posicion_camara_anterior = camara_principal.worldPosition.copy()
    objetos_detectados_anteriores = objetos_detectados
```

Fragmento de código 37. Obtención de las velocidades a evitar en función de la orientación de las superficies de contacto.

Si en el módulo “control_camara_7.py” del fichero de ejemplo se activa la parte de código correspondiente y se lanza el juego se podrá observar que, aunque en general el comportamiento es correcto, aparecen los problemas previstos:

- La cámara no evita los choques contra paredes cuando se encuentra dentro de objetos, o en general cuando la pared contra la que choca no se encuentra entre ella y el origen del objeto al que pertenece. Como ejemplo vale el recinto circular que se ve de frente al lanzar el juego, o cualquiera de las rampas presentes en el escenario.

- Si choca contra un objeto grande, de manera que el origen esté lejos, es posible que el rayo entre al objeto por una cara distinta a la que ha chocado contra la cámara. Si dicha cara tiene una orientación diferente, el cálculo de la normal será erróneo. Este problema es similar al producido cuando un objeto ajeno se interpone en el camino del rayo, excepto porque este último se puede evitar indicando al método "rayCast" que compruebe si el objeto atravesado tiene la propiedad establecida previamente en el que detectó el sensor "Near". Sin embargo en este caso ambas caras pertenecen al mismo objeto, por lo que no es posible razonar de esa forma. En la práctica se puede observar haciendo chocar a la cámara con alguno de los objetos que se han preparado expresamente para ello, como una superficie horizontal alargada con un pico hacia arriba en su parte central.

Como se puede observar, todos los problemas, los que se han podido atajar fácilmente y los que no, tienen su origen en la imposibilidad de conocer el punto exacto en que se produce el contacto entre la esfera de detección del sensor "Near" y el objeto ajeno.

5.9.4. Toma de decisiones basada en el lanzamiento de varios rayos

Ante los problemas que genera el no poder conocer el punto de detección del objeto que choca, se ha considerado otro planteamiento diferente: mientras el sensor "Near" detecte contacto, trazar un rayo hacia cada una de las posibles direcciones (y sentidos) en que se pueden aplicar las velocidades: derecha, izquierda, arriba, abajo, y atrás (es posible dejar de tratar la velocidad lineal por separado); si alguno de estos rayos choca contra un objeto, en el cuadro presente se evitarán velocidades en su dirección y sentido.

Los rayos se pueden trazar mediante "rayCast" o "rayCastTo", ya que solamente es necesario saber si han chocado o no, y su longitud debe ser similar al rango de detección del sensor "Near".

Al usar este método, al contrario que con el anterior, no es viable tomar decisiones cuando se produce un cambio en la lista de objetos detectados por el sensor "Near", ya que en la mayoría de las ocasiones dicho cambio no implica que alguno de los rayos atraviese o deje de atravesar algún objeto. Como ejemplo de esto se puede observar la Figura 9, donde se ha representado tanto la esfera de detección del sensor "Near" como cada rayo lanzado por "rayCast", siendo su longitud igual al radio de la esfera. Aunque el sensor "Near" ha detectado la cercanía de la pared, ninguno de los rayos llega a tocarla, con lo cual no se parará ninguna velocidad, cuando lo correcto sería hacer lo propio con la lateral y la lineal.

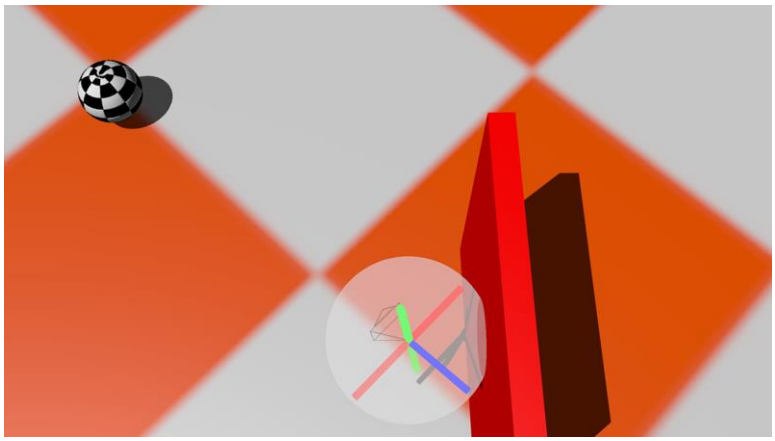


Figura 9. Relación entre la esfera de detección del sensor “Near” y los rayos trazados por “rayCast”.

Por tanto es necesario comprobar, en cada cuadro y durante todo el tiempo que el sensor “Near” esté activo, la presencia de objetos en cada dirección. El uso del sensor podría omitirse, y trazar rayos continuamente, pero ello supondría un gasto innecesario de capacidad de procesamiento.

A continuación se detallan los pasos a seguir a nivel de código, que se reflejan en el Fragmento de código 38. El mismo código está presente en el módulo “control_camara_7.py” del fichero de ejemplo (se denomina “Determinar velocidades a parar – lanzamiento de varios rayos”), y para usarlo se deberá primeramente activar, quitando las comillas triples que lo encierran, y anular el correspondiente al método anterior (todo el código desde su título, “Determinar velocidades a parar – orientación de superficies”, hasta el comienzo del bloque actual). También es necesario eliminar la asignación de la variable global “objetos_detectados_anteriores”, en la parte inferior, ya que de lo contrario se produciría un error, al no existir “objetos_detectados”.

- En primer lugar se deben declarar las variables “parar_velocidad_lateral”, “parar_velocidad_vertical”, y “parar_velocidad_lineal”, dándoles el valor 0. Debido a que se actualizan en todos aquellos cuadros en los que sean susceptibles de tomar valores distintos de 0 (cuando el sensor “Near” detecte se active), no es necesario que sean globales, por lo que las declaraciones del nivel superior y del principio de la función “main” se podrían eliminar. En todo caso sí es imperativo darles el valor 0 antes de la búsqueda de obstáculos, para que si ésta es negativa no se evite ningún movimiento.
- Después se comprueba si se cumplen las condiciones que hacen necesario parar alguna velocidad: que el sensor detecte cercanía de algún objeto (“pared_cerca” sea “True”) y no haya obstáculos. Como esta lógica se ejecuta en cada cuadro, al contrario que al usar la normal, se pueden descartar directamente los casos donde la pelota no sea visible (los choques contra el suelo se tratarán más aparte).
- Si se cumplen ambas condiciones se procede a “lanzar” los rayos correspondientes. Como en principio los sentidos opuestos (arriba/abajo, derecha/izquierda) son

excluyentes, se puede comprobar primero uno de ellos y, si el resultado es negativo, el otro. Dado que solamente es necesario distinguir si el rayo choca, o no, contra cualquier objeto, se usa el método "rayCastTo" (ver apartado 5.5.2, página 64). Para cada comprobación se pasa como parámetro el punto del espacio que se encuentra a una distancia determinada (por ejemplo el radio de detección del sensor "Near") de la cámara en la dirección y sentido correspondiente. La dirección se obtiene de las variables calculadas al principio del módulo ("direccion_velocidad_lateral", "direccion_velocidad_vertical", "direccion_velocidad_lineal"), y el sentido se establece mediante el signo de estos vectores de dirección. En el Fragmento de código 38 el orden en que se lanzan los rayos es: izquierda, derecha, arriba, abajo, atrás.

- Si, en alguna de las comprobaciones anteriores, "rayCastTo" devuelve un objeto (no devuelve "None"), se pararán la velocidad y sentido correspondientes según de qué rayo se trate. En el caso de la velocidad lineal el valor que se dé a "parar_velocidad_lineal" es indiferente, ya que en la decisión de restringirla o no sólo se comprueba si dicha variable es nula o tiene algún valor.
- Por último, si entre la cámara y el protagonista había obstáculos, con lo cual no se realizan las comprobaciones anteriores, pero el sensor "Near" sí está activo, y el suelo está entre los objetos detectados por este sensor, se debe parar la velocidad vertical en sentido negativo. Aunque la cámara esté detrás de la pared, y por tanto se le permita atravesarla, no debe poder hacer lo mismo con el suelo.

```
# Asignaciones del nivel superior...

def main():

parar_velocidad_lateral,parar_velocidad_vertical,parar_velocidad_linea
l = 0,0,0
    if pared_cerca and not hay_obstaculo:
        if camara_principal.rayCastTo(camara_principal.worldPosition -
2 * direccion_velocidad_lateral):
            parar_velocidad_lateral = -1
        elif camara_principal.rayCastTo(camara_principal.worldPosition
+ 2 * direccion_velocidad_lateral):
            parar_velocidad_lateral = 1
        if camara_principal.rayCastTo(camara_principal.worldPosition +
2 * direccion_velocidad_vertical):
            parar_velocidad_vertical = 1
        elif camara_principal.rayCastTo(camara_principal.worldPosition
- 2 * direccion_velocidad_vertical):
            parar_velocidad_vertical = -1
        if camara_principal.rayCastTo(camara_principal.worldPosition -
2 * direccion_velocidad_lineal):
            parar_velocidad_lineal = 1
    elif pared_cerca and 'suelo' in sensor_pared_cerca.hitObjectList:
            parar_velocidad_vertical = -1
```

Fragmento de código 38. Obtención de las velocidades a evitar basándose en el lanzamiento de rayos en cinco direcciones.

Como se puede comprobar al ejecutar el fichero de ejemplo con este bloque de código activado, con este sistema se solucionan todos los problemas que presentaba el método anterior, y la simplicidad en cuanto al código es mayor.

5.9.5. Detención total del desplazamiento vertical

Hasta este punto, siempre que se ha actuado sobre la velocidad vertical ha aparecido el problema, previsible según lo comentado en el apartado 5.3.1 (página 51), de que la cámara no se deja de mover en vertical aunque la velocidad correspondiente sea nula. Mientras que al restringir el ángulo de inclinación de la cámara este hecho era inapreciable, en el tratamiento de obstáculos su efecto es importante, ya que puede hacer que se lleguen a atravesar superficies, por ejemplo el suelo, o producir vibraciones molestas cuando el obstáculo está encima de la cámara.

La solución pasa por, cuando se deba "parar" la velocidad vertical, sustituir el valor nulo de "velocidad_vertical_final" por otro que haga que la cámara no se mueva en vertical. Para ello se usa un razonamiento basado en la lógica que mantiene la posición vertical inicial, explicada en el apartado 5.3.1. En este caso se modifica dicha lógica para que la posición a mantener sea, en lugar de la inicial, la que tiene la cámara en el momento del contacto.

En cuanto al código, se debe añadir un bloque más, y una variable global que almacene dicha posición (llamada en el ejemplo "altura_camara_momento_choque"). Esta variable valdrá "None" cuando no se deba parar la velocidad vertical, y guardará la altura a mantener mientras perdure el contacto que ocasionó la restricción de la velocidad vertical. A continuación se explican los pasos a seguir, que se corresponden con el código mostrado en el Fragmento de código 39:

- Si la velocidad vertical final es cero, es decir, la combinación de la variable "parar_velocidad_vertical" y la velocidad vertical calculada hacen que no se pueda aplicar dicha velocidad, se entra en el bloque lógico para sustituir el cero por el valor adecuado.
- En éste, en primer lugar se decide si se debe actualizar la variable que contiene la altura objetivo, en función de su propio valor anterior: si era nulo ("None" ó 0), el actual es el primer cuadro en que se restringe la velocidad vertical, y por tanto se debe almacenar la altura actual de la cámara, que perdurará mientras se mantenga el contacto con el obstáculo; si el valor anterior no era nulo se debe respetar.
- Posteriormente se sustituye el valor de "velocidad_vertical_final", que es nulo, por el necesario para mantener la posición dada, usando el mismo razonamiento que el explicado en el citado apartado 5.3.1.
- Por último, en caso de que "velocidad_vertical_final" no sea cero, es decir, no haya que parar la velocidad vertical, se debe anular la altura objetivo para, en ocasiones posteriores, saber que hay que refrescar su valor. Para no hacer esta operación

continuamente, su ejecución se condiciona además a que el valor anterior de la variable no fuera nulo: si ya lo era, no es necesario repetir la asignación.

Además es necesario realizar un cambio fundamental en el sistema que evita que la velocidad vertical se pare en seco (ver apartado 5.8.1, página 82). Este sistema se basa en la velocidad que se aplicó a la cámara en el cuadro anterior, y asume que es nula cuando la cámara no se mueve, lo cual no es cierto en el caso del movimiento en vertical. Por tanto, la asignación de la variable que almacena la velocidad anterior ("velocidad_vertical_anterior" en este caso) debe hacerse antes que la corrección para mantener la posición, de manera que su valor sea nulo cuando la cámara esté parada.

Si no se cambiase la línea correspondiente, la variable "velocidad_vertical_anterior" contendría un valor positivo siempre que se restringiese la velocidad vertical; entre el cuadro en que se deja de mover el ratón y el cuadro en que empieza a actuar el sistema que mantiene la posición vertical inicial, la velocidad calculada es cero, por lo que, al ser la anterior positiva, el bloque de código que evita las paradas bruscas la sustituiría por una positiva pero menor que la anterior, lo que provocaría que la cámara descendiese lentamente.

En el Fragmento de código 39 aparece el código a añadir para que la cámara mantenga realmente su posición vertical cuando se requiera, así como la nueva localización de la línea que actualiza la variable "velocidad_vertical_anterior". En el módulo "control_camara_7.py" del fichero de ejemplo se deben quitar las comillas triples del bloque de código correspondiente (llamado "Calcular la velocidad vertical necesaria para mantener la posición vertical") para activarlo, y anular la asignación de "velocidad_vertical_anterior" de la parte inferior. La nueva asignación de esta variable se encuentra al principio de dicho bloque.

```
# Resto de asignaciones...

altura_camara_momento_choque = 0

def main():
    global altura_camara_momento_choque
    # Resto de lógica...

    velocidad_vertical_anterior = velocidad_vertical_final

    if velocidad_vertical_final == 0:
        if not altura_camara_momento_choque:
            altura_camara_momento_choque =
camara_principal.position[2]
        altura_camara_actual = camara_principal.position[2]
        velocidad_vertical_final = 10 * (altura_camara_momento_choque
- altura_camara_actual)
        direccion_velocidad_vertical = Vector([0,0,1])

    elif altura_camara_momento_choque:
        altura_camara_momento_choque = None
```

Fragmento de código 39. Código a añadir para mantener la cámara a una altura constante cuando se deba detener su velocidad vertical.

5.9.6. Detención de la velocidad en ambos sentidos de una misma dirección

El tratamiento de los choques implementado anteriormente puede resultar insuficiente en casos donde la cámara se mueve entre paredes muy próximas, concretamente si la distancia entre las paredes es menor que el diámetro de la esfera de detección del sensor "Near". Entonces, dado que hasta ahora los sentidos en que se para la velocidad son excluyentes (si se impide el movimiento hacia arriba no se hace hacia abajo, ídem en cuanto a izquierda y derecha), se permite que la cámara atraviese una de las paredes (si se usa el método de la normal, mientras que si se trazan rayos en varias direcciones puede suceder bien que se atraviesen las paredes o bien que se produzcan movimientos indeseables).

Si esto se desea evitar basta con añadir la siguiente lógica: si en primer lugar se determina que se debe parar una velocidad en un sentido, y en el siguiente ciclo del bucle (método de la normal), o al lanzar el siguiente rayo se debe parar el contrario, en lugar de dar a la variable correspondiente el valor de esta última comprobación (como se viene haciendo hasta ahora) se le asigna uno distinto, por ejemplo el número 2.

También se deben modificar los bloques de código que razonan si se debe permitir el movimiento en cada dirección para actuar en consecuencia cuando las variables "parar_velocidad_lateral" o "parar_velocidad_vertical" valgan 2. En esos casos se debe evitar la velocidad correspondiente siempre, ya que el movimiento en ambos sentidos implica acercarse hacia una pared.

En el Fragmento de código 40 aparece el código que es necesario añadir/modificar en cuanto a la velocidad lateral, siendo equivalente lo relativo a la vertical. Aunque se ha partido del último método explicado (lanzamiento de varios rayos), el razonamiento es similar para el método de la normal. En el fichero de ejemplo, para evitar una complejidad excesiva, se ha incluido un nuevo módulo ("control_camara_8.py") solamente para reflejar estos cambios. El resto de código es igual que el en el módulo anterior, aunque se han eliminado los bloques que estaban comentados. Para probarlo se debe por tanto configurar el controlador "Python" dedicado a la cámara para que lance la función "main" de dicho módulo ("control_camara_8.main").

```
# Asignaciones del nivel superior...

def main():
    if pared_cerca and not hay_obstaculo:
        if camara_principal.rayCastTo(camara_principal.worldPosition -
2 * direccion_velocidad_lateral):
            parar_velocidad_lateral = -1
        if camara_principal.rayCastTo(camara_principal.worldPosition +
2 * direccion_velocidad_lateral):
            if not parar_velocidad_lateral:
                parar_velocidad_lateral = 1
            else:
                parar_velocidad_lateral = 2

    # Resto de lógica...

    if coinciden_signos(velocidad_lateral_final,
parar_velocidad_lateral) or parar_velocidad_lateral == 2:
        velocidad_lateral_final = 0
```

Fragmento de código 40. Detención de la velocidad en ambos sentidos de una misma dirección.

6. IMPLEMENTACIÓN Y CONTROL DEL MINI-MAPA DE ORIENTACIÓN

En este apartado se explican los pasos necesarios para que aparezca en la pantalla un mini-mapa que muestre esquemáticamente la situación del objeto protagonista en cada momento. En líneas generales lo que se hace es, mediante el módulo *Video Texture* ("bge.texture"), sustituir la textura de un objeto (en este caso un plano, llamado "mapa") por la imagen capturada por una cámara cenital, y actualizar dicha imagen continuamente.

Los diferentes puntos se han ordenado por complejidad, de manera que en primer lugar se explica cómo implementarlo de la manera más básica, para después añadir mejoras y funcionalidades gradualmente.

En el archivo "demo_control_mapa.blend" se han incluido diferentes *scripts* relacionados con las etapas de implementación que se irán comentando a continuación, llamados "control_mapa_1.py", control_mapa_2.py", etc. Cuando, en el texto, se haga referencia a un cambio de un *script*, bastará con cambiar en el controlador "Python" correspondiente (situado en el objeto "empty_escena_mapa" de la escena "escena_mapa") el módulo que lanza, llamando siempre a la función "main" (por ejemplo: "control_mapa_1.main").

6.1. Elementos necesarios

Para conseguir que aparezca información superpuesta en la pantalla, en este caso un plano con una textura dinámica, pueden seguirse dos estrategias:

- Colocarlo en la misma escena donde se desarrolla la acción, muy próximo a la cámara principal y emparentado con ella, de manera que desde su punto de vista el plano esté delante de todos los demás objetos. Este método, si bien es el menos complejo, tiene algunos inconvenientes: la cámara podría en un momento dado situarse a una distancia menor de un objeto que la que le separa del mapa, en cuyo caso el mapa dejaría de verse; por otra parte, cualquier cambio en los parámetros de la cámara requeriría re-ajustar el tamaño y posición del objeto "mapa" y, si se quisieran colocar más cámaras para ver el juego desde diferentes puntos de vista, habría que colocar un nuevo plano junto a cada una de ellas.
- Colocarlo en una nueva escena, que se superpone a la principal al iniciarse la ejecución. Aunque es algo más complicado de configurar, por la necesidad de trabajar en dos escenas diferentes, este método se considera más apropiado ya que, además de resolver todos los problemas relacionados con el método anterior, establece una separación conceptual entre la acción principal y lo relacionado con el mapa.

A la vista de las opciones, es más conveniente colocar el objeto "mapa" en una nueva escena (llamada "escena_mapa"). Dicha escena se crea desde cero, y debe contener tres objetos únicamente: el plano que contendrá el mapa (llamado "mapa"), la cámara que lo enfoca ("camara_mapa"), y el *Empty* correspondiente desde el que se lanza la lógica ("empty_escena_mapa"). Este *Empty* podría omitirse, ejecutando las funciones

correspondientes desde la escena principal, pero de esa manera se perdería la separación entre ambas áreas y se perjudicaría la inteligibilidad, al haber una gran cantidad de controladores asociados al objeto "juego" de la escena principal.

A continuación aparece la relación de todos los objetos que intervienen en la implementación del mini-mapa, clasificados por escenas:

- "escena_principal" (donde se desarrolla la acción):
    - "camara_mapa": cámara que captura la acción desde arriba. La imagen que genera se convertirá en la textura del objeto "mapa".
    - Copia de los objetos del mapa: copia de todos los objetos que intervienen en el juego y sus posiciones relativas, colocada debajo de dichos objetos (con una desviación de -1000 unidades en el eje Z). Sus nombres son los mismos que los de los objetos que representan pero terminados en "_mapa".
- "escena_mapa" (se superpondrá a la escena principal durante la ejecución):
    - "mapa": objeto tipo plano cuya textura será sustituida durante la ejecución para mostrar lo captado por "camara_mapa".
    - "camara_escena_mapa": cámara que enfoca únicamente al objeto "mapa". La imagen que genera se superpondrá completamente a la escena principal durante la ejecución del juego, por lo que el mapa sólo debe ocupar una parte pequeña de su campo de visión.
    - "empty_escena_mapa": colocado en el centro de la escena, su única función es almacenar y lanzar la lógica.

6.2. Creación y configuración de los elementos

A continuación se trata detalladamente el proceso a seguir para crear y configurar cada uno de los elementos anteriores. Todo lo explicado a continuación se refleja en el fichero de referencia "demo_control_mapa.blend", que es conveniente ir consultando al mismo tiempo.

6.2.1. Cámara "camara_mapa"

Es simplemente una cámara que apunta en la dirección negativa del eje Z, o en otras palabras "mira hacia abajo", para captar las imágenes que aparecerán en el mapa. Debe situarse por tanto en la escena principal, en la vertical del objeto protagonista y apuntando hacia éste.

La lente se configura como ortográfica ("Orthographic"), de manera que usa una perspectiva ortogonal en lugar de cónica, como ocurre si el modo es "Perspective". En un mini-mapa no es deseable, en general, la perspectiva cónica (la que se conoce comúnmente como perspectiva a secas), ya que podrían producirse efectos negativos, el más evidente que un objeto alto oculte a otros más bajos. En perspectiva ortogonal no existe el concepto de punto de vista, si no que todos los puntos se proyectan sobre el plano de visión paralelamente

a la dirección en la que apunta la cámara; esto implica que sólo será visible la parte superior de los objetos (lo que se busca), pero también que la altura a la que se coloca la cámara no influye en la cantidad de objetos que entran en el campo de visión ni en su tamaño. Puede comprobarse cómo moverla en la dirección del eje Z no cambia la imagen que capta. El parámetro que controla el tamaño del plano de visión (efecto *zoom*) es el llamado "Orthographic Scale".

Por lo general existe la necesidad de que la cámara vaya siguiendo al objeto protagonista: de otra manera, excepto en escenarios muy pequeños, la cámara tendría que tener un parámetro "Orthographic Scale" muy grande, es decir, un *zoom* muy pequeño, para abarcar la totalidad de la escena, de manera que los objetos aparecerían muy pequeños (probablemente indistinguibles) y el mapa no cumpliría su función de ayudar a la orientación. Por tanto, para dicho parámetro se elige un valor de compromiso que permita identificar al protagonista y los objetos que le rodean (para hacer este ajuste es conveniente estar viendo al mismo tiempo la imagen que entrega la cámara, seleccionándola y pulsando CONTROL + teclado numérico 0). El seguimiento se consigue en un principio emparentando la cámara al objeto protagonista (en modo "Vertex", para que afecte a la posición pero no a la rotación, ya que la pelota gira libremente). En apartados posteriores se estudiará la manera de efectuar el seguimiento de forma más intuitiva, rotando la cámara y modificando su *zoom* en función de los movimientos del protagonista.

6.2.2. Copia de los objetos del juego

Consiste en una copia exacta, en cuanto a posición relativa, escala y rotación, de todos los objetos (a excepción de cámaras, luces y *empties*), colocada por debajo de los objetos "originales" en la misma escena "escena_principal", por motivos que se explican más adelante. Esta copia no se utiliza en la implementación más básica del mini-mapa, sino que será la base de las mejoras visuales que se introducen en el apartado 6.4 (página 119), al proporcionar la independencia entre el mini-mapa y la escena necesaria para poder eliminar sombras y texturas, introducir un icono en lugar de la pelota y controlar ambos por separado.

Se ha decidido que, para facilitar su manejo desde Python, y favorecer la propia función del mini mapa, los objetos copiados deben presentar las siguientes características:

- En cuanto a su forma y colocación relativa son una copia exacta de todos los que intervienen en el juego, situada en la misma escena ("escena_principal"). Su posición absoluta está desviada 1000 unidades de medida de Blender en la dirección negativa del eje Z respecto a los objetos "originales". Es decir, la copia de una pared cuyas coordenadas XYZ sean, por ejemplo, [5,10,2], se colocará en [5,10,-998] (ver Figura 10).
- El nombre de dichas copias consiste en el del objeto al que representan más la terminación "_mapa" añadida al final.
- El material es de tipo "Shadeless" en todos los casos, ya que no es deseable que la luz afecte al mini-mapa. El color "Diffuse" es el mismo que el del material del primer *slot*

de materiales del objeto al que representa. Los casos especiales, como el del objeto protagonista, se pueden tratar a parte.

- El tipo físico de todas las copias es “Static”. Las copias que representan a objetos cuyo tipo físico es distinto de “Static” tienen una propiedad de juego llamada “dinamico” con valor “1”.

La decisión de colocar la copia en la misma escena, que rompe con la idea de separación de conceptos (por la cual debería colocarse en la escena “escena_mapa” o en otra nueva) ha estado condicionada por el *script* que se ha diseñado para realizarla, explicado a continuación. Colocar las copias en una escena diferente requiere una complejidad mucho mayor, y se ha considerado que ni el pequeño aumento en comodidad que proporciona ni el mantener la coherencia con el resto de la organización lo justifican. En la práctica, el único inconveniente derivado de colocar los objetos nuevos en la misma escena es la necesidad de “bajar” hasta ellos en caso de que fuera necesario hacer algún retoque, lo cual se puede atajar usando dos ventanas simultáneamente, una para modificar los objetos originales y la otra para hacerlo con las copias.

La decisión de colocar la copia 1000 unidades por debajo ha sido relativamente arbitraria: se ha puesto a una distancia suficiente como para que no haya interferencias entre objetos de ambos escenarios y que la copia quede fuera del alcance de visión de la cámara “camara_principal”, pero no demasiado grande para minimizar la incomodidad comentada anteriormente.

En la Figura 10 aparece una captura de pantalla de Blender donde se muestra una posible posición relativa entre los objetos originales y sus copias.

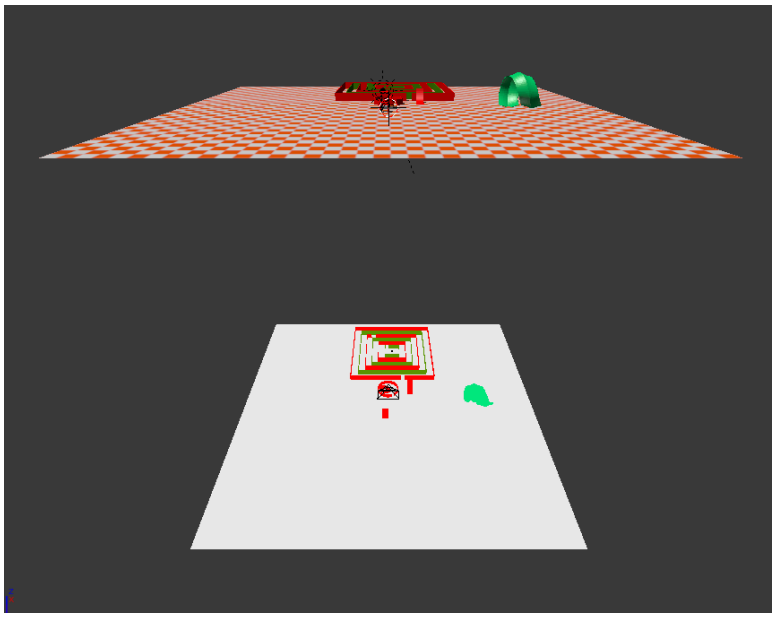


Figura 10. Posición de los objetos del juego (arriba) y sus copias (abajo), usadas en el mini-mapa.

Como se ha comentado, durante la realización de este proyecto, dado que la réplica en sí es un proceso tedioso y propenso a causar errores, se decidió diseñar un *script* (llamado "crear_mapa.py") que la lleva a cabo automáticamente. Aunque dicho *script* queda fuera del ámbito del proyecto, ya que se basa en los módulos de la aplicación (ver apartado 3.3, página 20), a continuación se explican brevemente las acciones que realiza, y el código completo se ha incluido en el Anexo I.

- En primer lugar se importan los módulos necesarios, y se crean e inicializan las variables. Los nombres de la escena y el objeto protagonista, así como la altura a la que se colocan las copias de los objetos, se definen como variables de manera que el *script* pueda ser adaptado con facilidad a otros escenarios o necesidades.
- Posteriormente se eliminan todos los objetos relacionados con el mapa que pudieran existir anteriormente (se distinguen comprobando si su nombre termina en "_mapa"). Este paso permite que el *script* pueda usarse además para actualizar el mapa, cuando se añadan, quiten o modifiquen objetos que intervienen en el juego. Aunque probablemente podrían usarse técnicas que actualizaran sólo los objetos necesarios en lugar de eliminar todos y volver a crearlos, al no ser éste el objetivo del proyecto, y dado que el tiempo empleado en realizar la copia no se considera importante (es del orden de décimas de segundo en el fichero de ejemplo), se ha optado por seguir la vía más simple y directa.
- Después se entra en un bucle "for" que recorre todos los objetos de la escena, y se comprueba si tienen asociado un material, en cuyo caso se copiarán (de esta manera se evita copiar objetos innecesarios o incluso perjudiciales, como lámparas, *empties* o cámaras). Si corresponde duplicarlos, se hacen las siguientes operaciones:
    - Se crea un nuevo objeto, cuyo nombre es el mismo que el del objeto original con la terminación "_mapa", y su estructura es una copia de la del objeto original.
    - El objeto recién creado se asocia a la escena "escena_principal".
    - La escala y rotación del nuevo objeto se igualan a las del original. La localización se iguala la del original más el desvío vertical correspondiente (variable definida al principio del *script*). Si el desvío es de 1000 unidades, la copia de una pared cuyas coordenadas XYZ sean, por ejemplo, [5,10,2], se colocará en [5,10,-998] (ver Figura 10).
    - A aquellos que representan a objetos susceptibles de moverse durante el juego se les añade una propiedad, que se usará posteriormente para saber que deben ser actualizados en cada cuadro.
    - Se eliminan todos los materiales del objeto nuevo, y se le asigna uno solo, del mismo color que la propiedad "Diffuse" del primer material del objeto original, y de tipo "Shadeless", para que no se vea afectado por la luz, lo cual no es deseable en un mapa. El material que se asigna debe crearse previamente

o, si ya existía, recuperado de la lista de materiales. La comprobación de si un material existe o es necesario crearlo se hace basándose en su nombre, que contiene las componentes de su color. Como la creación/recuperación de materiales es necesaria en otros puntos del *script*, se ha colocado dentro de la función "obtener_material". Esta función toma como parámetro un color y busca en la lista de materiales si existe alguno con el nombre correspondiente; si es así devuelve una referencia al mismo, en caso contrario lo crea y devuelve la referencia correspondiente.

- A continuación se crea la cámara que captará las imágenes, configurando su lente como "Orthographic". Aunque su posición se controlará en tiempo real, y por tanto no es importante colocarla en ningún punto en concreto, se hace en el origen de coordenadas y a cierta altura sobre el resto del mapa.
- Por último se sustituye el objeto que representa al protagonista (una esfera en este caso) por otro que, visto desde arriba, indique con mayor claridad su posición y dirección, como por ejemplo un cono rotado convenientemente. Se le asigna un material de un color diferente del resto de los presentes en el mapa, de forma que se distinga con claridad (este material se crea aprovechando la función mencionada anteriormente).

6.2.3. Plano "mapa"

Es el plano cuya textura se sustituirá por la imagen que entrega la cámara "camara_mapa". Se coloca en una escena aparte ("escena_mapa"), y es enfocado por la cámara "camara_escena_mapa", cuya salida se superpondrá a la escena principal durante la ejecución del juego. A continuación se enumeran las características que debe tener dicho plano:

- Posición: cualquiera, aunque se prefiere colocar paralelo a alguno de los planos de referencia (en este caso el XY) para que posteriormente sea más sencillo orientar la cámara.
- Relación de aspecto: la misma que la del juego (determinada en la pestaña "render"), en este caso 16:9, para que la imagen no aparezca distorsionada. Las dimensiones no afectan al funcionamiento siempre que las relaciones de aspecto coincidan.
- Material: cualquiera (en este caso se ha dejado el que se crea por defecto) con la opción "shadeless" activada, de manera que no se vea afectado por la luz, y consecuentemente no sea necesario añadir una lámpara a la escena.
- Textura: de tipo "Image or Movie". Aunque se puede usar cualquier imagen, ya que será sustituida al arrancar el juego, se prefiere generar una directamente desde Blender (ver Figura 11), de la misma relación de aspecto que el objeto "mapa" (en este caso 16:9) y rellena con una rejilla UV (*Generated Type: UV Grid*); de esta manera es posible detectar errores en la relación de aspecto del plano. Una vez creada y guardada

la imagen con un nombre representativo (por ejemplo “textura_mapa.png”), se asigna a la textura del objeto, y se establece el mapeo con coordenadas UV. Al entrar en modo edición, hacer *unwrap* y ajustar los vértices del objeto a los de la imagen (preferiblemente usando sólo la función de escalar, para mantener las proporciones), el resultado en la ventana “3D View” debe ser similar al de la Figura 12, es decir, los cuadrados de la rejilla no deben aparecer distorsionados. Esto indica que la relación de aspecto del objeto “mapa” es correcta. El número de cuadrados o su tamaño no tienen por qué coincidir, ya que dependen de las dimensiones absolutas de la imagen y del plano, parámetros que no son determinantes siempre que las relaciones de aspecto sean correctas.

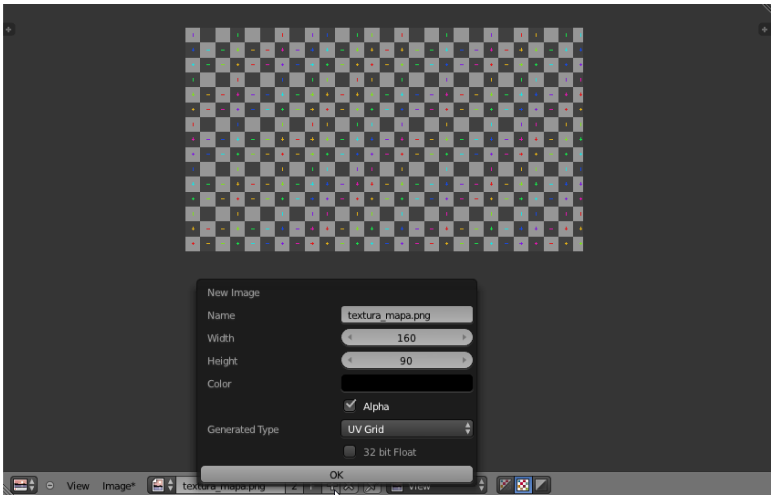


Figura 11. Creación de una textura con relación de aspecto 16:9 y rejilla UV.

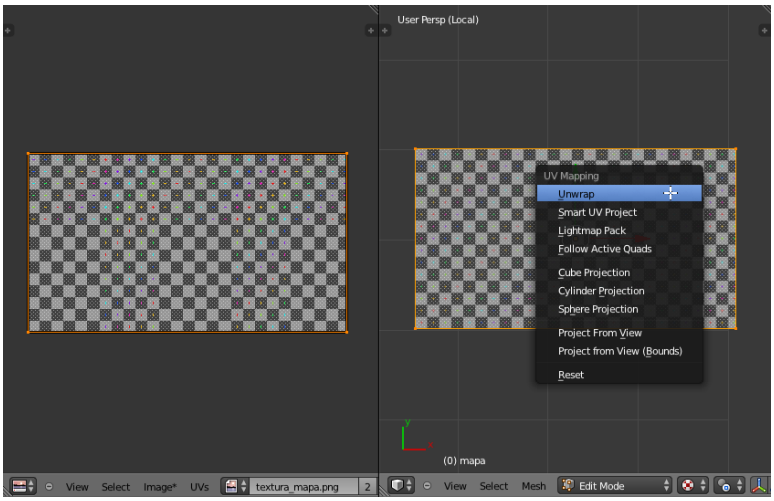


Figura 12. Ventanas “UV/Image Editor” y “3D View” tras un *unwrapping* y un ajuste correcto.

6.2.4. Cámara "camara_escena_mapa"

Es la cámara cuya salida se superpondrá a la escena principal durante el juego. El único requisito es que su eje Z se coloque en la dirección perpendicular al plano "mapa", para que no se produzca distorsión, es decir, que la cámara apunte perpendicularmente al plano.

Si se cumple la condición anterior, y dado que el plano es un objeto de dos dimensiones, no habrá diferencia entre usar lentes "Orthographic" o "Perspective". En el archivo de demostración se ha configurado como "Orthographic" para mantener el paralelismo con la cámara "camara_mapa".

Hay que tener en cuenta que la imagen que entrega esta cámara se superpondrá a toda la pantalla cuando comience el juego, por tanto no se puede hacer que el plano ocupe todo el área de visión, en cuyo caso al ejecutar el juego el resto de la acción quedaría oculta por él. Para ajustar su tamaño y posición lo más conveniente es activar la vista de la cámara (teclado numérico 0), y moverlo y escalarlo hasta que quede como se desee.

6.2.5. *Empty* "empty_escena_mapa"

Consiste en un objeto tipo "Empty" colocado en cualquier lugar de la escena "escena_mapa", en este caso en el centro, cuya única finalidad es contener los *logic bricks* necesarios para ejecutar la lógica que controla el mapa.

El uso de este *empty* es completamente prescindible, por una parte porque las mismas funciones que se llaman desde él podrían ejecutarse desde el *empty* de la escena principal, y por otra porque los *logic bricks* que contiene podrían asociarse a otro objeto de la escena (la cámara o el plano). Sin embargo se ha hecho así para mantener la coherencia con las decisiones de organización tomadas (ver apartado 3.6), que tienen la ventaja de separar la lógica perteneciente a diferentes escenas, colocando cada *script* en la escena sobre la que actúa, y evitar buscar en qué objeto se encuentran los *logic bricks*.

En la Figura 13 aparece una captura de pantalla de los elementos de la escena "escena_mapa". Nótese que las posiciones relativas entre la cámara y el plano serán diferentes en función del tipo de lente que se escoja, de sus parámetros y de las dimensiones absolutas del plano.

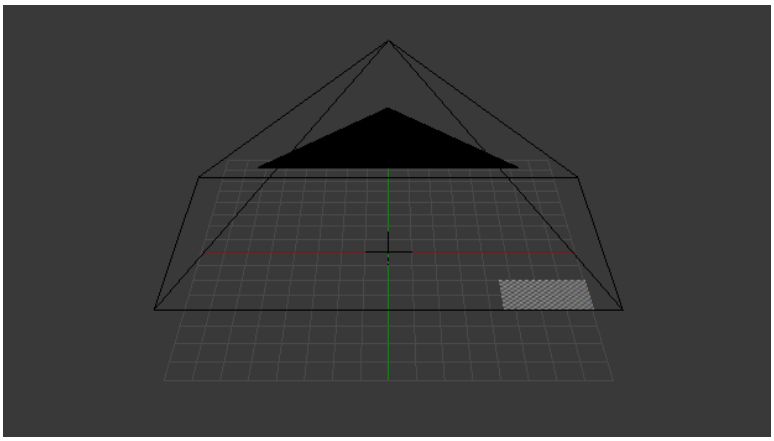


Figura 13. Elementos de la escena “escena_mapa”: “camara_escena_mapa”, “mapa”, “empty_escena_mapa”.

6.3. Implementación de la versión más básica del mini-mapa

En este apartado se explican los pasos necesarios para simplemente conseguir que a la imagen del juego se superponga el plano “mapa”, y que se sustituya la textura de éste por una vista de la escena desde arriba. En el fichero de ejemplo, “demo_mapa.blend”, el módulo correspondiente a esta primera fase de implementación del mapa es el “control_mapa_1.py”. Recuérdese que los *logic bricks* correspondientes se encuentran asociados al objeto “empty_escena_mapa” de la escena “escena_mapa”.

6.3.1. Superposición de la escena

Consiste en añadir la escena al principio de la ejecución del juego, mediante el método “addScene(name,overlay=1)” del módulo “bge.logic”. El parámetro “name” es la denominación que se ha dado a la escena a añadir al crearla en Blender, en este caso “escena_mapa”, y “overlay” es una variable booleana opcional que indica si la escena se añade en el fondo (si vale 0) o al frente (si vale 1 o se omite). En la Figura 14 se incluye una comparación entre ambos modos de funcionamiento (la escena que se añade sólo contiene un plano azul rectangular y la cámara correspondiente). Por tanto en este caso el comando a utilizar será: `logic.addScene('escena_mapa',1).`

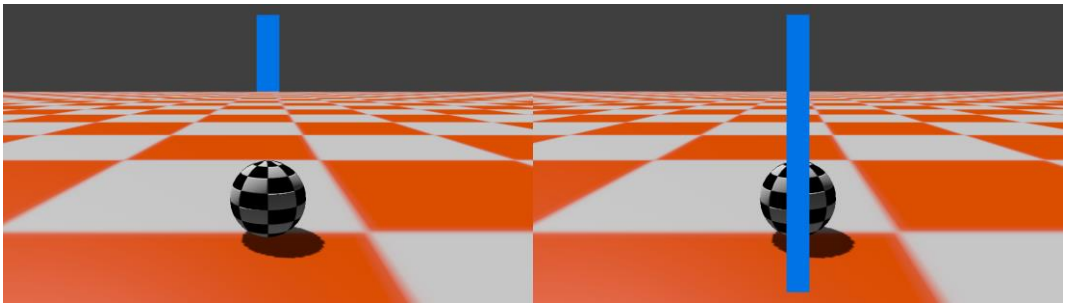


Figura 14. Comparación entre el resultado del método “addScene” con el parámetro “overlay” a 0 (izquierda) y a 1 (derecha).

El código anterior debe lanzarse desde un *logic brick* perteneciente a la escena principal, que es desde donde se inicia el juego y consecuentemente la única que existe en el primer cuadro. La lógica perteneciente a otras escenas no se ejecuta hasta que no sean añadidas, momento en que se ejecutarán por primera vez los módulos y por tanto se inicializarán las variables y se definirán las funciones.

Dado que, en la escena principal, sólo es necesaria la línea de código anterior desde el punto de vista del mini-mapa, dicha línea se incluye en el módulo "logica_comun.py", que contiene bien funciones relacionadas con varias áreas a la vez o bien funciones que, como es el caso, son necesarias en la escena principal pero su finalidad no está relacionada con ninguno de los otros bloques (por ejemplo no tiene que ver con el control de la cámara, ni con el de la pelota).

En el Fragmento de código 41 aparece el código necesario para añadir la escena al principio de la ejecución, tal y como está escrito en el módulo "lógica_comun.py" del fichero de referencia. Nótese que de dicho módulo se ha eliminado todo el código ajeno al mini-mapa y se ha colocado en otro diferente, para mostrar solamente lo relacionado con este apartado. Dado que la escena se añade una sola vez al comenzar el juego, el código correspondiente se coloca en el nivel superior, y la función "main()", al haber eliminado el resto del código, queda vacía, por lo que es necesario añadir la palabra clave "pass".

```
from bge import logic

logic.addScene('escena_mapa',1)

def main():
    pass
```

Fragmento de código 41. Código necesario para superponer una escena al principio de la ejecución. Corresponde al módulo "lógica_comun.py" de la escena "escena_principal".

Si se dedicase un controlador "Python" solamente a añadir la escena lo más lógico sería configurarlo en modo "Script" y conectarlo a un sensor "Always" con "pulse mode" desactivado, de manera que se ejecutara una sola vez al comenzar el juego. En ese caso, aunque no imprescindible ya que no produce errores, sería conveniente comprobar si el sensor se encuentra en el pulso positivo, para no añadir la escena una vez está presente.

Para comprobar que la escena se superpone correctamente basta con lanzar el juego (P) desde la escena principal; debe aparecer un rectángulo en la posición que se haya colocado al configurar la escena "escena_mapa", en este caso en la parte inferior derecha de la pantalla, con la textura que se le haya asignado, la rejilla UV si se ha seguido el apartado anterior.

6.3.2. Sustitución de la textura

Una vez se consigue que la escena se superponga al ejecutar el juego, sólo resta sustituir la textura del rectángulo por la imagen entregada por la cámara "camara_mapa". La lógica dedicada a ello se sitúa en el *empty* "empty_escena_mapa" de la escena "escena_mapa"; en él

solamente es necesario añadir un sensor “Always” con “Pulse Mode” activado, y conectarlo a un controlador “Python” en modo “Module”, desde el que se lanza el módulo “control_mapa.py”.

El núcleo de la sustitución es un objeto de tipo “Texture” (definido en el módulo “texture”) que toma el control sobre una textura existente, sustituyéndola por otra imagen que a su vez puede proceder de diferentes sitios. En resumen, consiste en configurar en dicho objeto qué textura debe sustituir, qué imagen debe poner en su lugar, y cuándo tiene que hacerlo.

El primer paso es crear una instancia de la clase “Texture”, llamada en este caso “nueva_textura”; al hacerlo es necesario pasar como parámetro el objeto cuya textura se quiere cambiar, en este caso llamado “mapa”. Esto es válido en los casos simples, donde el objeto tiene un solo material y una sola textura. Si tuviera más de un material haría falta pasar como parámetro el identificador del material que contiene la textura, y si el material estuviera asociado a más de una textura habría que pasar también el número de *slot* de la textura a modificar. Para obtener el identificador del material se usaría la función “materialID” contenida en el módulo “texture”, pasándole como parámetros el objeto en que se encuentra el material y el nombre de la textura que se quiere cambiar. En el Fragmento de código 42 se incluye tanto el método simple como, entre comentarios, el que usa “materialID”.

Posteriormente se debe establecer qué imagen debe sustituir a la textura original. La textura nueva puede proceder de distintas fuentes, como un vídeo, un archivo de imagen, o, como es el caso, una cámara de la escena. La función a usar en cada situación para obtener la imagen correspondiente se puede consultar en la sección “Video Texture” de la documentación de Blender; en este caso es “ImageRender”, que toma como parámetros una cámara y la escena a la que pertenece y devuelve la imagen captada por dicha cámara.

Es necesario por tanto disponer previamente de una referencia a la escena “escena_principal” y a uno de sus objetos, la cámara “camara_mapa”. La primera se consigue recorriendo la lista de las escenas activas hasta encontrar una cuyo nombre coincide con el de la buscada (“escena_principal”), y la segunda accediendo directamente a la lista de objetos de la escena anterior, usando como índice el nombre de la cámara (“camara_mapa”). Ver Fragmento de código 42 o módulo “control_mapa_1.py” en el fichero de ejemplo.

Para configurar el objeto “nueva_textura” de forma que sustituya la textura original del objeto “mapa” por la imagen captada por la cámara “camara_mapa”, es necesario asociar su atributo “source” a la imagen devuelta por la función “ImageRender” cuando se le pasan como parámetros las dos referencias anteriores. La línea de código correspondiente queda: `nueva_textura.source = ImageRender(escena_principal ,camara_mapa).`

Por defecto Blender asigna al fondo (zona donde la cámara “camara_mapa” no capta ningún objeto) el color azul. Para modificarlo se debe cambiar el atributo “background” de la fuente de la siguiente manera: `nueva_textura.source.background = [100,100,100,255]`. El color está representado en RGBA (RGB más canal alfa) de 8 bits (0-255), por lo que la línea anterior corresponde a un color gris.

Toda la lógica anterior se coloca en el nivel superior del módulo “control_mapa.py”, y por tanto se ejecuta una sola vez, en el momento en que se añade la escena “escena_mapa” y se crea el objeto que la contiene (“empty_escena_mapa”). Sin embargo, para que el mini-mapa se actualice en tiempo real es necesario, en cada cuadro, llamar al método “refresh” del objeto “nueva_textura”; lo que hace este método es eliminar la imagen cargada en el atributo “source”, de manera que en el cuadro siguiente se detecte que falta y se tome de nuevo la imagen captada por la cámara en ese momento. Para comprobar este aspecto se puede por ejemplo colocar la llamada a “refresh” en un bloque condicional, de manera que sólo se ejecute cada cierto número de cuadros; según dicha cifra sea mayor la frecuencia de actualización será menor. El método “refresh” requiere un parámetro de tipo “Bool”, necesario pero cuyo valor no influye en el resultado (a fecha de la última edición de este texto la documentación de Blender no refleja cuál es el por qué ni la finalidad de dicho parámetro), por tanto en este caso se pasará el valor “True”. En total, dentro de la función “main”, que es la que lanza el controlador “Python” en cada cuadro, solamente es necesario escribir: `nueva_textura.refresh(True).`

En el Fragmento de código 42 se incluye todo el código necesario para llevar a cabo la sustitución de texturas, que junto con lo explicado en apartados anteriores lleva a la obtención de un mini-mapa muy básico.

```
from bge import logic, texture

# Variables relacionadas con la escena actual ("escena_mapa")
escena_mapa = logic.getCurrentScene()
mapa = escena_mapa.objects['mapa']

# Variables relacionadas con la escena principal
for escena in logic.getSceneList():
    if escena.name == 'escena_principal':
        escena_principal = escena
        break
camara_mapa = escena_principal.objects['camara_mapa']

# Metodo simple
nueva_textura = texture.Texture(mapa)

# Metodo avanzado, no necesario en este caso
#id_material = texture.materialID(mapa,'IMtextura_mapa.png')
#nueva_textura = texture.Texture(mapa,id_material,0)

nueva_textura.source =
texture.ImageRender(escena_principal,camara_mapa)
nueva_textura.source.background = [100,100,100,255]

def main():
    nueva_textura.refresh(True)
```

Fragmento de código 42. Código necesario para sustituir la textura original por una dinámica en el objeto “mapa”. Corresponde al módulo “control_mapa.py” de la escena “escena_mapa”.

6.4. Mejoras visuales

Los pasos llevados a cabo en los apartados anteriores conducen simplemente a superponer en la pantalla de juego un rectángulo donde aparece la acción vista desde otra perspectiva (desde arriba). En esta sección se explicarán algunas modificaciones que se pueden implementar para mejorar el resultado visual de manera que dicho rectángulo se acerque algo más a los mini-mapas que normalmente aparecen en los juegos comerciales. El módulo del fichero de ejemplo ("demo_control_mapa.blend") correspondiente a este apartado es el llamado "control_mapa_2.py", a configurar en el objeto "empty_escena_mapa" de la escena "escena_mapa" (haciendo que lance "control_mapa_2.main"). En dicho módulo, los diferentes bloques de código que a los que se hace referencia en este apartado están en un principio anulados (entre comillas triples), de manera que puedan ir activándose gradualmente según se avanza en el texto, y así observar el efecto de cada uno.

En la Figura 15 se muestra una comparación entre la apariencia actual del mini-mapa y la que se pretende conseguir. Como se puede observar, hay una serie de diferencias que benefician la claridad e inteligibilidad:

- Se han eliminado las sombras. Quizá el punto más importante; la función de las sombras es añadir realismo y ayudar a la orientación cuando son vistas en la pantalla principal, sin embargo en el mini-mapa estas aplicaciones no tienen sentido, e incluso provocan confusión. Es más, en caso de existiera algún objeto en la escena del mismo color (gris oscuro) que las sombras, no sería posible distinguir su contorno de las mismas.
- Se ha eliminado la textura del suelo. Desde el punto de vista del jugador dicha textura es vital, como se comentó, para dar sensación de movimiento; sin embargo, dado que el mapa aparece superpuesto a la escena principal, esa pista visual no es necesaria. Además, de nuevo pueden darse casos donde sea contraproducente: en un escenario donde hubiese una carretera y a los lados paredes u objetos blancos, las líneas pintadas en el asfalto ocasionarían confusión.
- Se ha sustituido el objeto protagonista por un icono que lo representa. De nuevo la textura en el mini-mapa es superflua, ya que su finalidad es intervenir en la pantalla principal, en el punto de vista del jugador. También hay que tener en cuenta que el mini-mapa sólo ocupa una porción de la pantalla principal, y en él el objeto protagonista se ve desde una distancia mayor, con lo cual aparecerá muy pequeño, haciendo que la textura sea indistinguible. Además, el icono por el que se ha sustituido tiene forma de triángulo isósceles, lo que permite señalar la dirección en la que se desplaza.

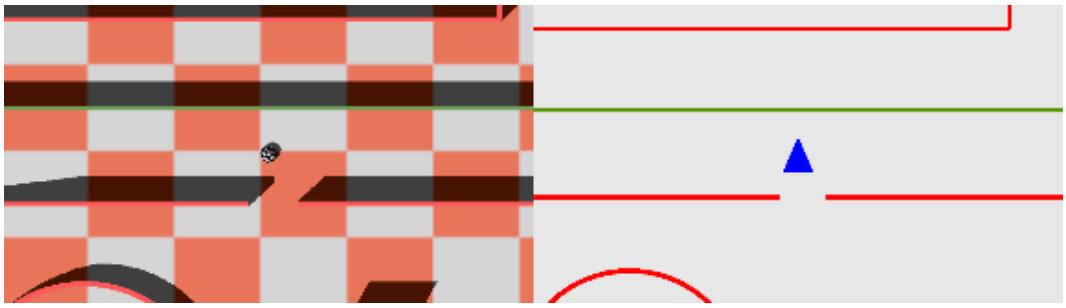


Figura 15. Apariencia del mini-mapa antes (izquierda) y después (derecha) de introducir las mejoras visuales.

Para conseguir estas características es necesario hacer algunas modificaciones en la configuración descrita en los apartados anteriores, que se detallan a continuación.

6.4.1. Copia de los objetos del juego

El cambio más importante, respecto al apartado anterior, necesario para implementar estas mejoras, es hacer que la cámara "camara_mapa" enfoque a otros objetos diferentes a los que aparecen en el escenario del juego: a una copia de los mismos. De esta manera se pueden controlar por separado sus apariencias y comportamientos.

Los detalles sobre la copia aparecen en el apartado 6.2.2 (página 109). Para ejecutar este *script* basta con visualizarlo en el editor de texto y presionar el botón "Run Script". Es conveniente guardar antes el progreso, ya errores inesperados podrían hacer que Blender se cerrase de improviso. Una vez ejecutado, se podrá observar cómo se han creado las copias de los objetos por debajo de los ya existentes.

Para los apartados posteriores se debe tener en cuenta que el nombre de todas estas copias termina en "_mapa", que el objeto que representa a la pelota se llama "icono_mapa", y que los objetos cuya posición puede cambiar y deben ser actualizados tienen una propiedad del juego llamada "dinamico".

6.4.2. Sustitución y movimiento básico de la cámara "camara_mapa"

La creación de una copia de los objetos requiere hacer cambios en la cámara "camara_mapa", de manera que enfoque a esta copia en lugar de a los objetos originales, como hacía hasta ahora. Aunque una posibilidad sería desplazar hacia abajo la cámara existente, si se ha ejecutado el *script* "crear_mapa.py" se habrá eliminado automáticamente la cámara antigua y se habrá creado una nueva en la posición adecuada, con el mismo nombre, de tipo "Orthographic", apuntando hacia abajo y con un valor "Orthographic Scale" (equivalente al zoom) de 30.

Si se lanza el juego, en la ventana del mini-mapa aparecerán los nuevos objetos, aunque no habrá ningún tipo de respuesta a los movimientos del objeto protagonista, ya que de momento no se ha implementado.

En este caso se ha decidido controlar el movimiento de la cámara mediante Python, en lugar de emparentarla con la pelota como se hizo en la versión más básica. Esto se consigue

simplemente igualando las coordenadas globales X e Y de la posición de la cámara a las de la pelota. Para ello es necesario obtener una referencia al objeto "pelota", y añadir en la función "main" las asignaciones correspondientes. En el Fragmento de código 43 aparece el código a añadir al módulo con el que se terminó el apartado 6.3.

```
# Resto de asignaciones…

pelota = escena_principal.objects['pelota']


def main():
    # Resto de lógica…

    camara_mapa.worldPosition[0] = pelota.worldPosition[0]
    camara_mapa.worldPosition[1] = pelota.worldPosition[1]
```

Fragmento de código 43. Movimiento básico de la cámara "camara_mapa" siguiendo al objeto protagonista.

Al lanzar el juego y mover la pelota se observa cómo la imagen efectivamente sigue los movimientos de la misma, pero el icono (triángulo azul) que la representa permanece estático. El movimiento de dicho icono se trata en el apartado siguiente.

6.4.3. Movimiento básico del icono que representa al objeto protagonista

El desplazamiento sobre el plano horizontal se implementa de manera similar al de la cámara, tratado en el apartado anterior. Ya que el icono representa al objeto protagonista, su posición tendrá las mismas coordenadas (en el plano horizontal) que la de éste. En el módulo correspondiente basta con obtener una referencia al objeto "icono_mapa" y, en cada cuadro, igualar sus coordenadas X e Y a las de la pelota.

Si se lanza el juego en este estado se observará que ahora el icono se desplaza por el mapa (aparece estático en el centro del recuadro, ya que la cámara ocupa su misma posición). Este comportamiento podría ser suficiente, dado que indica la situación del objeto protagonista en el plano; sin embargo en este caso el mini-mapa se ha ideado para mostrar también su dirección, por lo que se ha usado un icono con forma de flecha. Si simplemente se quiere indicar la posición, es conveniente que el icono presente simetría esférica o radial. A continuación se explican los pasos necesarios para que el icono se oriente en la dirección de avance de la pelota.

En primer lugar es necesario conocer la dirección de avance de la pelota, para lo que se accede a la propiedad "direccion_pelota" del objeto "pelota". El proceso de obtención y almacenaje de dicho valor está explicado en el apartado 7.7(página 144).

Posteriormente, y teniendo en cuenta que la orientación de la "punta" (la parte que indica la dirección) del icono coincide con la de su eje X, sólo resta modificar convenientemente su atributo "worldOrientation" (en este caso, ya que el icono no es "hijo" de ningún otro objeto, serviría también "localOrientation"). El uso del mismo está explicado en el apartado 7.8 (página 147), que precisamente se apoya en este ejemplo. Como el vector "direccion_pelota"

está normalizado, basta con asignar directamente sus componentes a los dos primeros valores de la primera columna (eje X) de la matriz "worldOrientation", y las de un vector perpendicular a él hacia la izquierda a la segunda columna (eje Y). Dado que el giro se produce en el plano horizontal y que el eje Z del icono siempre apunta hacia el eje Z global, no es necesario modificar ninguno de los elementos de la matriz que hacen referencia este eje.

En el Fragmento de código 44 se incluye el código necesario para implementar el movimiento y la rotación del icono:

```
# Resto de asignaciones...

icono = escena_principal.objects['icono_mapa']

def main():
    # Resto de lógica...

    direccion_pelota = pelota['direccion_pelota']

    icono.worldPosition[0] = pelota.worldPosition[0]
    icono.worldPosition[1] = pelota.worldPosition[1]

    icono.worldOrientation[0][0] = direccion_pelota[0]
    icono.worldOrientation[1][0] = direccion_pelota[1]
    icono.worldOrientation[0][1] = direccion_pelota[1]
    icono.worldOrientation[1][1] = -direccion_pelota[0]
```

Fragmento de código 44. Movimiento y rotación del icono que representa al objeto protagonista.

Ejecutando el juego en este punto se debería obtener el comportamiento deseado: el icono siempre está en el centro del mini-mapa, e indica la dirección de la pelota. En apartados posteriores se explica cómo intentar evitar las brusquedades que se producen cuando la pelota choca con las paredes o hace cambios de sentido.

6.4.4. Actualización de los objetos

Al igual que es necesario controlar, usando Python, los movimientos y rotación del icono y de la cámara que enfoca al mapa, hay que hacer lo propio con las copias de aquellos objetos que sufran modificaciones durante el juego, para que éstas se reflejen en el mini-mapa.

La manera de proceder es simplemente modificar los atributos "worldPosition" y "worldOrientation" de los objetos del mapa para que coincidan con los de los objetos que representan, excepto por la tercera componente de "worldPosition", que debe tener en cuenta que el mapa tiene un desvío vertical (1000 unidades en este caso).

Dado que por lo general no todos los objetos de la escena serán móviles (en este caso sólo lo son una minoría), hacer dichas operaciones indiscriminadamente es inefectivo, y podría afectar notoriamente al rendimiento. Es necesario distinguir entre los objetos que nunca cambian su posición u orientación (tipo físico "Static") y los que sí pueden hacerlo, y sólo actuar sobre éstos últimos.

La forma que se ha considerado más conveniente para diferenciarlos ha sido añadir una propiedad del juego a las copias que representen a objetos susceptibles de moverse (tipo físico diferente de "Static"), llamada "dinamico" y con valor "1" (el valor es irrelevante). De esta manera la distinción, que implica también dejar de lado otros objetos como cámaras, lámparas o el objeto protagonista, ya está hecha antes de ejecutar el juego y se reduce la lógica necesaria. La propiedad se añade automáticamente durante la creación de la copia de los objetos (al ejecutar el *script* "crear_mapa.py").

En el módulo que controla el mapa, en este caso "control_mapa_2.py", el código correspondiente a la actualización del mapa se distribuye en dos partes. En el nivel superior, de manera que se ejecute una sola vez, se buscan los objetos del mapa que tienen la propiedad "dinamico" y aquellos a los que representan, de manera que posteriormente se puedan consultar las posiciones y orientaciones de los originales y modificar las de las copias. Las referencias a ellos se almacenan en la lista "objetos_dinamicos" por parejas: cada elemento de esta lista es a su vez otra lista donde en primer lugar se encuentra el objeto del mapa y en segundo el correspondiente objeto "original". Para encontrar estos últimos debe conocerse su nombre, que es posible deducir sabiendo que el nombre de las copias es el del objeto al que representan más la terminación "_mapa", usando el método "split".

El método "split", presente en todas las cadenas de caracteres, devuelve una lista de cadenas de caracteres compuesta por los fragmentos de la original que quedan a los lados de los caracteres que coinciden con el que se pasa por parámetro. Por ejemplo, el comando `'ab.cd.ef'.split('.')` devuelve la siguiente lista: `['ab','cd','ef']`. En el caso del mini-mapa, "split" se usa para obtener la parte del nombre de las copias de objetos que queda antes del guión bajo (_), es decir, el primer elemento de la lista devuelta.

Posteriormente, en una función que se ejecute continuamente, este caso "main", se debe llevar a cabo la actualización propiamente dicha. Consiste simplemente en recorrer la lista "objetos_dinamicos", asignando a cada objeto del mapa la posición (teniendo en cuenta el desvío vertical) y orientación que tiene en ese momento el objeto al que representa.

El código correspondiente a todo el proceso de actualización aparece en el Fragmento de código 45.

```
# Resto de asignaciones...

objetos_dinamicos = []
for objeto in escena_principal.objects:
    if 'objeto_dinamico' in objeto:
        nombre_original = objeto.name.split('_')[0]
        #nombre_original = '_'.join(objeto.name.split('_')[0:-1])
        objeto_original = escena_principal.objects[nombre_original]
        objetos_dinamicos.append([objeto,objeto_original])

DESVIACION_MAPA = 1000

def main():
    # Resto de lógica...

    for pareja_objetos in objetos_dinamicos:
        pareja_objetos[0].position[0] = pareja_objetos[1].position[0]
        pareja_objetos[0].position[1] = pareja_objetos[1].position[1]
        pareja_objetos[0].position[2] = pareja_objetos[1].position[2]
- DESVIACION_MAPA
        pareja_objetos[0].orientation = pareja_objetos[1].orientation
```

Fragmento de código 45. Actualización de los objetos del mapa.

La línea comentada constituye otra forma de obtener el nombre del objeto "original" a partir del de la copia, que, si bien no es necesaria en este caso, podría serlo si el nombre del objeto copia tuviera más de un guión bajo. Si por ejemplo existiese una pared llamada "pared_roja", su correspondiente copia se denominaría "pared_roja_mapa", y por tanto aplicando el método sencillo (el que aparece sin comentar) se obtendría como nombre original "pared", que es incorrecto. El funcionamiento de la línea comentada es el siguiente: el método "split" devolvería la lista `['pared','roja','mapa']`, de la cual se seleccionan los elementos del primero (0) al penúltimo (anterior a -1), con lo que quedaría `['pared','roja']`. Resta unir ambas cadenas colocando un guión bajo entre ellas, para lo cual la manera más fácil y efectiva es usar el método "join". Éste une las cadenas pasadas como parámetro colocando entre ellas el carácter (o cadena) desde el que ha sido llamado (si sólo se pasa una, la devuelve sin modificar). El resultado será por tanto "pared_roja".

6.4.5. Control dinámico del *zoom*

Probablemente el mayor inconveniente que presenta el mini-mapa obtenido en los apartados anteriores es que el *zoom* de la cámara se configura a priori (mediante su parámetro "Orthographic Scale"), y no cambia mientras el juego está corriendo. Esto obliga a adoptar un compromiso entre el nivel de detalle (deseado cuando se hacen movimientos pequeños) y el área cubierta por el mapa (se prefiere que sea grande cuando el protagonista se desplaza a gran velocidad).

Una solución a este problema es modificar el nivel de *zoom* en función de la velocidad del objeto protagonista, de manera que a baja velocidad haya mucho detalle, pero según se acelere la cámara se "aleje" y aumente la superficie visible. Para ello solamente es necesario acceder a las variables que contienen o definen ambas magnitudes y relacionarlas mediante una ecuación.

El efecto *zoom* de la cámara, como se ha indicado en el apartado 6.2.1, está gobernado por su parámetro "Orthographic Scale", al tratarse de una cámara ortográfica. Su valor se consulta o modifica accediendo al atributo "ortho_scale" del objeto "camara_mapa".

En cuanto a la velocidad de la pelota, lo más sencillo es obtenerla del atributo "worldLinearVelocity", presente en todos los objetos del juego. Este atributo contiene un vector tridimensional con las tres componentes de la velocidad del objeto en cada momento. Como no es deseable que el *zoom* reaccione a los movimientos verticales (si lo hiciera, cambiaría por ejemplo al saltar), solo se tienen en cuenta las dos primeras componentes, correspondientes al plano horizontal. La manera más rápida de conocer el dato de interés, la velocidad absoluta, es crear un nuevo vector con estas dos componentes y consultar el atributo "length", que contiene su módulo. En el *script* se ha denominado "velocidad_pelota".

La relación entre esta velocidad y la escala de la cámara se establece de la siguiente manera: en vez de hacer depender la escala únicamente de la velocidad, se define como la suma de la escala que tiene inicialmente y un valor que depende de la velocidad. De esta forma es posible ajustar el valor base (el *zoom* cuando la pelota está parada) directamente en el visor 3D. Para ello se define una variable en el nivel superior que contenga dicha escala inicial, en este caso llamada "escala_camara_inicial", y en cada cuadro se calcula la nueva escala ("escala_camara_deseada") sumándole una cantidad que depende de la velocidad de la pelota (en concreto se suma el valor de la velocidad multiplicado por un una constante, "SENSIBILIDAD_ZOOM", definida en el nivel superior). Ver Fragmento de código 46.

```
# Resto de asignaciones...

escala_camara_inicial = camara_mapa.ortho_scale

SENSIBILIDAD_ZOOM = 3

def main():
    # Resto de lógica...

    velocidad_pelota = Vector(pelota.worldLinearVelocity[0:2]).length

    escala_camara_deseada = escala_camara_inicial + SENSIBILIDAD_ZOOM
* velocidad_pelota
    camara_mapa.ortho_scale = escala_camara_deseada
```

Fragmento de código 46. Control del *zoom* de la cámara en función de la velocidad de la pelota.

Si se añaden estos cambios a los realizados anteriormente se observará cómo efectivamente la cámara "se aleja" según la pelota acelera. Sin embargo aparece también la consecuencia indeseada de que cambios bruscos en la velocidad (que desde el punto de vista del jugador no se perciben como tal, por ejemplo un choque lateral con una pared, o pasar por escalones) ocasionan "saltos" de *zoom* muy notorios y molestos. Asimismo, a la vez que el resto de los objetos aparecen más pequeños según se aumenta la velocidad, el icono que representa al objeto protagonista hace lo mismo, pudiendo llegar al extremo de ser indistinguible. A continuación se explican posibles soluciones para ambos problemas.

Para suavizar los cambios bruscos de zoom se usa el método explicado en la última parte del apartado 7.6: en lugar de introducir directamente el valor calculado ("escala_camara_deseada"), se aplica una fracción de la diferencia entre dicho valor y el que "escala_camara" tiene actualmente. La fracción de la diferencia, que determina la rapidez con la que se alcanzarán los valores que debería tomar la variable si no se usara este método se controla en este caso mediante la constante "SUAVIDAD_ZOOM", definida en el nivel superior. Su valor podrá oscilar entre 0 (el nivel de *zoom* no cambia) y 1 (el nivel de *zoom* cambia inmediatamente). Si estos valores resultan anti intuitivos, por ser los más pequeños los que más suavidad ocasionan, la fracción puede definirse como "1- SUAVIDAD_ZOOM". En el Fragmento de código 47 aparecen las líneas que habría que añadir al módulo para conseguir este suavizado.

```
# Resto de asignaciones...

SUAVIDAD_ZOOM = 0.1

def main():
    # Resto de lógica...

    escala_camara_actual = camara_mapa.ortho_scale

    escala_camara_deseada = escala_camara_inicial + SENSIBILIDAD_ZOOM
* velocidad_pelota
    escala_camara_final = escala_camara_actual + SUAVIDAD_ZOOM *
(escala_camara_deseada - escala_camara_actual)
    camara_mapa.ortho_scale = escala_camara_final
```

Fragmento de código 47. Control del *zoom* de la cámara suavizando los cambios bruscos.

En cuanto al tamaño del icono, la solución es aplicarle, en cada momento, un factor de escala que compense los cambios en el *zoom* de la cámara. Consiste en hacer que entre la escala que tiene el icono en cada momento y la que tiene al principio haya la misma relación que entre el valor en cada instante y el inicial de "Orthographic Scale" de la cámara. Con lo cual en un primer paso se calcula el cociente entre "escala_camara_final" y "escala_camara_inicial", y posteriormente se multiplica el resultado ("relacion_escalas") por "escala_icono_inicial" para obtener el valor a aplicar.

La información sobre la escala de los objetos está almacenada en sus atributos "localScale" y "worldScale", que en este caso serán equivalentes, dado que el icono no tiene ninguna relación de parentesco. Como de estos dos solamente "localScale" es modificable desde un *script*, se trabajará con el mismo. Estos atributos son vectores tridimensionales pero, dado que el plano vertical no interviene en el mini-mapa, se trabaja sólo con las dos primeras componentes. Para poder multiplicar todos los elementos de un vector por un número directamente (escribiendo "`vector_resultado = vector_origen * número`"), es necesario que esté representado por un objeto de clase "Vector", ya que Python no permite multiplicar una lista de números por un número.

```
# Resto de asignaciones...

escala_icono_inicial = Vector(icono.localScale[0:2])

def main():
    # Resto de lógica...

    relacion_escalas = escala_camara_final/escala_camara_inicial;
    escala_icono_deseada = escala_icono_inicial * relacion_escalas;
    icono.localScale[0:2] = escala_icono_deseada
```

Fragmento de código 48. Escalado del icono que representa al objeto protagonista para que su tamaño aparente no varíe al cambiar el zoom de la cámara.

Combinando lo explicado hasta ahora se tiene un mini-mapa que cumple el objetivo principal, es decir, representar con claridad la posición del objeto protagonista en el mapa.

6.5. Aspectos avanzados

En este apartado se describen otras formas más complejas de control del mini-mapa, así como la manera de mejorar algunos aspectos que pueden resultar molestos. Las explicaciones se reflejan en el módulo “control_mapa_3.py”, presente en el fichero de ejemplo. Para usarlo se debe configurar el controlador “Python” del objeto “empty_escena_mapa” de la escena “escena_mapa” de manera que lance “control_mapa_3.main”. Asimismo, al ir estudiando los diferentes apartados se deben activar los bloques de código correspondientes, que en un principio están encerrados entre comentarios.

6.5.1. Suavizado del giro del icono

Como se comprobó en el apartado 6.4.3 (movimiento básico del icono, página 121), el hecho de establecer la orientación del icono en función de la dirección de la pelota implica que, si se hace directamente, en el mini-mapa se ven cambios bruscos que en juego, en tercera persona, no lo parecen; el más evidente es el cambio de sentido: si se hace avanzar a la pelota hacia adelante, se para, y se mueve hacia atrás, el icono pasará de apuntar en un sentido a, en el instante siguiente, hacerlo en el contrario. Lo mismo ocurre en los choques contra paredes.

Aunque este comportamiento es teóricamente ideal, ya que representa la realidad, puede ser preferible evitar las brusquedades para conseguir una sensación global de suavidad. En este apartado se explican los pasos a seguir para ello.

El único método viable de los que se han estudiado está basado en el segundo de los explicados en el apartado 7.6 (suavizar movimientos, página 143). Consiste resumidamente en, en vez de apuntar el icono hacia la misma dirección que lleva la pelota, hacerlo hacia una intermedia, de manera que en cada cuadro la dirección en la que apunta el icono se va aproximando a la ideal.

La complejidad reside en la naturaleza de la magnitud que se pretende suavizar: no es posible hacer una operación del tipo: `direccion_nueva = direccion_antigua + FACTOR * (direccion_nueva - direccion_antigua)`, ya que no existe el concepto de suma o

producto de direcciones, y hacer estas operaciones directamente con las componentes de los vectores que las representan tampoco es viable.

Es necesario identificar las direcciones de la pelota y el icono por el ángulo que forman con un vector de referencia, de manera que sí es posible calcular un ángulo intermedio con facilidad. Posteriormente se hace la operación inversa, convirtiendo el ángulo resultado en un vector de dirección, y se aplica a la matriz "worldOrientation" del icono. En este caso se ha tomado como tal el eje X, es decir el vector [1,0] (nótese que en todo lo referente al mini-mapa no se usa el plano vertical).

A continuación se detallan los pasos del proceso:

- Obtención de los vectores de dirección sobre el plano horizontal de la pelota y el icono: la dirección de la pelota se obtiene directamente de su propiedad "direccion_pelota" (ver apartado 7.7, página 144). La del icono, es decir, la dirección en la que apunta su eje X, se toma de su matriz "worldOrientation" (ver apartado 6.4.3, página 121). Ambas direcciones se almacenan en objetos de clase "Vector"; el correspondiente al icono se denomina "direccion_icono_actual", y el que contiene la dirección de la pelota "direccion_icono_deseada" (se pretende que el icono apunte en la dirección de avance de la pelota). Ver Fragmento de código 49.
- Obtención de los ángulos que forman las direcciones con el eje X. Para ello se usa la función "obtener_angulo_x" (explicada en el apartado 7.9.3, página 152), que habrá que importar en el nivel superior desde el módulo "funciones_comunes.py". En este paso se obtienen las variables "angulo_icono_actual" y "angulo_icono_deseado".
- Cálculo de la diferencia entre los ángulos correspondientes a la dirección deseada para el icono y a la actual ("diferencia_angulos_icono"). Consiste en hacer la operación: `angulo_icono_deseado - angulo_icono_actual`, usando para ello la función "obtener_angulo_diferencia" del módulo "funciones_comunes.py" (ver apartado 7.9.4, página 153).
- Cálculo del ángulo definitivo que adoptará el icono ("angulo_icono_final"): se lleva a cabo sumando, al ángulo actual del icono ("angulo_icono_actual"), una fracción del ángulo diferencia calculado anteriormente ("diferencia_angulos_icono"). Dicha fracción está definida por la constante "RAPIDEZ_GIRO_ICONO", definida en el nivel superior.
- Obtención del vector de dirección correspondiente al ángulo anterior ("direccion_icono_final"). Sabiendo el ángulo que dicho vector forma con el eje X, sus componentes X e Y consisten respectivamente en el coseno y seno del ángulo.
- Modificación de la matriz "worldOrientation" para colocar el icono en la dirección deseada: el proceso es el mismo que el explicado en el apartado 6.4.3, en este caso usando "direccion_icono_final" en lugar de "direccion_pelota".

A continuación se incluye todo el código que interviene en este proceso:

```
# Resto de asignaciones...

RAPIDEZ_GIRO_ICONO = 0.1

def main():
    # Resto de lógica...

    direccion_icono_actual =
Vector([icono.worldOrientation[0][0],icono.worldOrientation[1][0]])
    direccion_icono_deseada = pelota['direccion_pelota']

    angulo_icono_actual = obtener_angulo_x(direccion_icono_actual)
    angulo_icono_deseado = obtener_angulo_x(direccion_icono_deseada)

    diferencia_angulos_icono = angulo_icono_deseado -
angulo_icono_actual

    if diferencia_angulos_icono > pi:
        diferencia_angulos_icono -= 2*pi
    if diferencia_angulos_icono < -pi:
        diferencia_angulos_icono += 2*pi

    angulo_icono_final = angulo_icono_actual + RAPIDEZ_GIRO_ICONO *
diferencia_angulos_icono

    direccion_icono_final =
[cos(angulo_icono_final),sin(angulo_icono_final)]

    icono.worldPosition[0] = pelota.worldPosition[0]
    icono.worldPosition[1] = pelota.worldPosition[1]

    icono.worldOrientation[0][0] = direccion_icono_final[0]
    icono.worldOrientation[1][0] = direccion_icono_final[1]
    icono.worldOrientation[0][1] = direccion_icono_final[1]
    icono.worldOrientation[1][1] = -direccion_icono_final[0]
```

Fragmento de código 49. Suavizado del giro del icono.

6.5.2. Orientación de la cámara “camara_mapa” en la dirección de avance de la pelota

Hasta este punto, la cámara del mini-mapa (“camara_mapa”) ha conservado siempre la misma orientación. En este apartado y en el siguiente se estudian otros modos de funcionamiento en los que la cámara rota en el plano horizontal en función de determinadas variables del juego. En el módulo “control_mapa_3.py” del fichero de ejemplo hay tres bloques de código correspondientes a estos dos apartados, desactivados en un principio mediante comillas triples. Cuando se quiera usar alguno es necesario editar las comillas de forma que sólo uno de los tres esté activo.

En este apartado concretamente, se hace girar a la cámara para colocarse en la dirección en la que avanza la pelota. Esto supone que la cámara gira solidariamente con el icono, y por tanto éste aparecerá siempre en el centro del mapa y sin moverse. En este caso es especialmente importante que los giros sean suaves, ya que en casos de cambios rápidos de dirección toda la ventana del mini-mapa sufriría brusquedades indeseables que ocasionarían distracciones de la acción principal.

En todos los casos donde se modifique o consulte la matriz “worldOrientation” de la cámara hay que tener en cuenta la colocación de sus ejes: como se aprecia en la Figura 16, donde aparecen las posiciones y orientaciones relativas entre la cámara “camara_mapa” y el icono “icono_mapa”, el eje de la cámara que se debe hacer coincidir con la dirección de la pelota no es el X sino el Y.

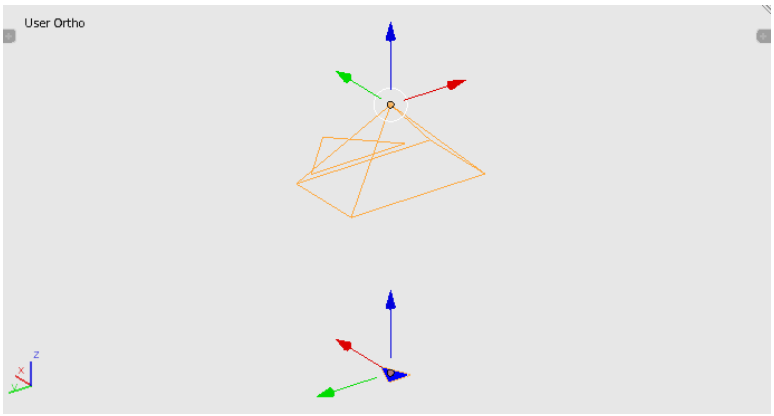


Figura 16. Orientación de los ejes de la cámara “camara_mapa” y el icono “icono_mapa”.

En caso de implementar un giro completamente solidario con el icono, el proceso consiste simplemente en aplicar a la cámara sus mismos cambios de orientación, cambios que ya incluyen el suavizado correspondiente (apartado 6.5.1). Simplemente habrá que tener en cuenta la orientación distinta de los ejes a la hora de introducir la dirección en la matriz “worldOrientation” de la cámara. La dirección deseada (“direccion_camara_final”) se debe colocar en la segunda columna de la matriz, correspondiente al eje Y. En la primera columna (eje X) se debe colocar un vector perpendicular a ésta hacia la derecha.

En el Fragmento de código 50 aparece el código que hace girar a la cámara solidariamente con el icono. La variable “direccion_camara_final” solamente se define para mantener la coherencia a la hora de hacer las asignaciones.

```
# Asignaciones del nivel superior...

def main():
    # Resto de lógica...

    direccion_camara_final = direccion_icono_final

    camara_mapa.worldOrientation[0][1] = direccion_camara_final[0]
    camara_mapa.worldOrientation[1][1] = direccion_camara_final[1]
    camara_mapa.worldOrientation[0][0] = direccion_camara_final[1]
    camara_mapa.worldOrientation[1][0] = -direccion_camara_final[0]
```

Fragmento de código 50. Giro de la cámara “camara_mapa” solidariamente con el icono “icono_mapa”.

Una variante de esta modalidad consiste en hacer que la cámara y el icono adapten su orientación a la dirección de la pelota con velocidades distintas. El resultado es que el icono

se ladea hacia la dirección en que se está girando, en mayor o menor medida según sea la diferencia entre lo rápido que la cámara y el icono adoptan la nueva orientación.

Esta rapidez está controlada, como se ha visto, por la constante "RAPIDEZ_GIRO_ICONO" en el caso del icono, y paralelamente por "RAPIDEZ_GIRO_CAMARA" en el de la cámara. Para poder tratar a cámara e icono de forma diferente es necesario calcular el proceso de suavizado del giro independientemente para cada uno de ellos. El único elemento en común es la dirección objetivo, la dirección que adoptarían si no se suavizasen los movimientos, que es la que lleva la pelota. El proceso de obtención de la dirección a aplicar a la cámara es exactamente igual al que se usa para el icono, explicado en el apartado 6.5.1.

En el Fragmento de código 51 se incluye el código correspondiente al cálculo y aplicación de la dirección que debe adoptar la cámara en cada cuadro.

```
# Resto de asignaciones...

RAPIDEZ_GIRO_CAMARA = 0.05

def main():
    # Resto de lógica...

    direccion_camara_actual =
Vector([camara_mapa.worldOrientation[0][1],camara_mapa.worldOrientatio
n[1][1]])
    direccion_camara_deseada = direccion_pelota

    angulo_camara_actual = obtener_angulo_x(direccion_camara_actual)
    angulo_camara_deseado = obtener_angulo_x(direccion_camara_deseada)

    diferencia_angulos_camara = angulo_camara_deseado -
angulo_camara_actual
    if diferencia_angulos_camara > pi:
        diferencia_angulos_camara -= 2*pi
    if diferencia_angulos_camara < -pi:
        diferencia_angulos_camara += 2*pi

    angulo_camara_final = angulo_camara_actual + RAPIDEZ_GIRO_CAMARA *
diferencia_angulos_camara
    direccion_camara_final =
[cos(angulo_camara_final),sin(angulo_camara_final)]

    camara_mapa.worldOrientation[0][0] = direccion_camara_final[1]
    camara_mapa.worldOrientation[1][0] = -direccion_camara_final[0]
    camara_mapa.worldOrientation[0][1] = direccion_camara_final[0]
    camara_mapa.worldOrientation[1][1] = direccion_camara_final[1]
```

Fragmento de código 51. Cálculo y aplicación del giro suavizado de la cámara "camara_mapa" en la dirección de la pelota.

6.5.3. Orientación de la cámara "camara_mapa" en la dirección en la que apunta la cámara "camara_principal"

Consiste en orientar la cámara "camara_mapa" de tal manera que, lo que en el mapa se entiende como su dirección de apunte (en realidad la cámara apunta hacia abajo, pero se considera que en este caso el mini-mapa apunta hacia la parte que queda arriba en la ventana,

que coincide con el eje Y de la cámara, ver Figura 16), coincida con la dirección en la que apunta la cámara principal, es decir, con la orientación opuesta a la del eje Z de la misma.

Dado que la cámara principal no apunta completamente en horizontal, y que en el mini-mapa sólo se considera este plano, al obtener la dirección en la que apunta el eje Z de su matriz "worldOrientation" (tercera columna) hay que eliminar lo relativo al plano vertical. Esto se hace tomando sólo sus dos primeras componentes y normalizando el resultado, como muestra el Fragmento de código 52.

Una vez se tiene el vector con la dirección de apunte de la cámara principal, que a su vez es la dirección a adoptar por la cámara del mini-mapa ("direccion_camara_final"), sólo resta modificar correctamente su matriz "worldOrientation". De nuevo debe tenerse en cuenta que hay que modificar tanto la columna correspondiente al eje Y (la segunda), como la correspondiente al X (la primera), colocando respectivamente el vector "direccion_camara_deseada" y uno perpendicular a él hacia la derecha.

En este caso, dado que la dirección a adoptar es la de apunte de la cámara principal, no es necesario aplicar suavizado: ya se ha hecho lo propio con los movimientos de dicha cámara en el módulo "control_camara.py". Además, cualquier movimiento brusco que pudiera darse en la cámara principal ocurriría al mismo tiempo en el mini-mapa, con lo que no constituirá una molestia mayor que la ocasionada en la propia pantalla principal.

En el Fragmento de código 52 aparecen las líneas del módulo "control_mapa_3.py" correspondientes a este método. Para usarlo en el fichero de prueba es necesario activarlo, quitando las comillas triples, y desactivar el anterior.

```
# Asignaciones del nivel superior...

def main():
    # Resto de lógica...

    direccion_camara_final = -
Vector([camara_principal.worldOrientation[0][2],camara_principal
.worldOrientation[1][2]])

    direccion_camara_final.normalize()

    camara_mapa.worldOrientation[0][0] = direccion_camara_final[1]
    camara_mapa.worldOrientation[1][0] = -direccion_camara_final[0]
    camara_mapa.worldOrientation[0][1] = direccion_camara_final[0]
    camara_mapa.worldOrientation[1][1] = direccion_camara_final[1]
```

Fragmento de código 52. Orientación de la cámara "camara_mapa" en la dirección de apunte de la cámara "camara_principal".

7. ASPECTOS COMUNES

En este apartado se explican las funciones o métodos de ámbito más general, que se han utilizado en más de un apartado o que no son específicas de uno de ellos en particular.

7.1. Comprobación del estado de una tecla o botón del ratón

Siempre que se necesite interactuar con teclas y botones habrá que manejar tres conceptos separados:

- El teclado y el ratón son objetos de clases predefinidas en Blender ("SCA_PythonKeyboard" y "SCA_PythonMouse" respectivamente), y como tal tienen una serie de atributos y métodos asociados. En este caso es interesante el atributo "events", presente en ambos. Dicho atributo es un diccionario que se actualiza constantemente, y que contiene parejas de valores, donde el primero es una constante que representa cada uno de los botones o teclas, y el segundo su estado en cada momento.
- En el módulo "events", incluido en el general "bge", se encuentran definidas las constantes que relacionan cada tecla (su nombre en lenguaje común) con el número de índice que le corresponde en el diccionario "events" (punto anterior). Por ejemplo, el retorno de carro, o "enter", está representado por la constante ENTERKEY, cuyo valor es concretamente 13. La lista completa de las constantes está publicada en la documentación de Blender.
- El segundo valor de cada pareja presente en "events" es el estado de los botones. Este campo puede tomar cuatro valores, también referenciados mediante constantes predefinidas, en este caso en el módulo "logic":
    - KX_INPUT_NONE: si la tecla no está pulsada. Corresponde al número 0.
    - KX_INPUT_JUST_ACTIVATED: si la tecla acaba de ser pulsada (primer cuadro en el que se detecta su pulsación). Corresponde al número 1.
    - KX_INPUT_ACTIVE: si la tecla está pulsada (y lo ha estado durante más de un cuadro). Corresponde al número 2.
    - KX_INPUT_JUST_RELEASED: si la tecla acaba de soltarse (primer cuadro en que deja de detectarse su pulsación). Corresponde al número 3.

Para demostrar con un ejemplo muy simple la manera de trabajar con las teclas (equivalente para los botones del ratón) se ha escrito el Fragmento de código 53, que imprime en pantalla un mensaje diferente en función del estado de la tecla "retorno de carro". En una aplicación real se incluiría el código a ejecutar en lugar del mensaje informativo.

```
from bge import logic, events

teclado = logic.keyboard
ACTIVO = logic.KX_INPUT_ACTIVE
RECIEN_ACTIVADO = logic.KX_INPUT_JUST_ACTIVATED
RECIEN_DESACTIVADO = logic.KX_INPUT_JUST_RELEASED

def main():
    if teclado.events[events.ENTERKEY] == ACTIVO:
        print('Tecla Enter pulsada')
    elif teclado.events[events.ENTERKEY] == RECIEN_ACTIVADO:
        print('Tecla Enter recien pulsada')
    elif teclado.events[events.ENTERKEY] == RECIEN_DESACTIVADO:
        print('Tecla Enter recien levantada')
```

Fragmento de código 53. Comprobación del estado de la tecla “retorno de carro”.

Nótese cómo sólo se han “traducido”, definiendo nuevas constantes, los posibles estados de una tecla, mientras que el número de la tecla en sí se ha referenciado mediante la constante predefinida en Blender. En este caso sí podría haberse escrito `TECLA_ENTER = events.ENTERKEY`, pero en un caso real no tiene sentido a menos que el número de teclas y botones que se usan sea muy reducido. De lo contrario, se acabaría teniendo un gran número de definiciones de constantes, que harían el código menos legible. Lo que sí es viable es asociar constantes a funciones específicas de la aplicación; por ejemplo, en el caso de este proyecto se hace `TECLA_PARAR = events.PKEY`, de manera que, si en un momento determinado se decide que la tecla destinada a parar la pelota debe ser otra, sólo se necesita cambiar una línea de código.

7.2. Uso de la clase “Vector”

La clase “Vector” está entre las más usadas a lo largo de este proyecto ya que, además de representar vectores como tal (lo que podría hacerse igualmente con listas de dos o tres elementos) incluye una serie de métodos que permiten hacer operaciones típicas en vectores de forma transparente. Para poder utilizarla es necesario importar su definición, que se encuentra en el módulo “mathutils”: `from mathutils import Vector`. A continuación se explica cómo crear objetos de clase “Vector”, el funcionamiento dichos métodos (los que se utilizan en este proyecto) y las operaciones manuales que se realizan con más frecuencia, y en el Fragmento de código 54 aparecen ejemplos prácticos de cada apartado:

- **Creación**: al crear un objeto de la clase “Vector” solamente es necesario pasar como parámetro una secuencia de números, que serán las componentes del vector. Esta secuencia se puede pasar indistintamente en forma de objeto tipo “list” o “tuple” (lo cual no influye en la posibilidad de editar los valores a posteriori). El número y valor de las componentes determina las operaciones que se pueden realizar con ellos; por ejemplo, no se puede calcular el ángulo que el vector [0,0,0] forma con otro cualquiera.

- **Operaciones aritméticas**: pueden realizarse directamente operaciones aritméticas entre vectores y con otros números, al contrario que si el vector se representa simplemente mediante una lista. Por ejemplo, multiplicar un número por un objeto "Vector" resulta en cada componente multiplicada por el número, mientras que hacer lo mismo con un objeto "list" devuelve un error.
- **Cálculo del módulo**: para obtener el módulo de un vector representado mediante un objeto "Vector" basta con acceder a su atributo "length".
- **Normalización**: consiste en obtener el vector unitario equivalente, es decir, un vector de módulo 1 con dirección y sentido equivalentes. En general es conveniente trabajar con vectores unitarios, y en muchos casos es necesario. El método "normalize" modifica el vector desde el que se le llama para hacerlo unitario, y el método "normalized" devuelve el vector desde el que se llamó normalizado. En este proyecto se usa únicamente el primero, mientras que el último es útil si se quiere conservar el vector original sin modificar.
- **Cálculo del ángulo formado con otro vector**: para llevarlo a cabo basta con llamar al método "angle" en uno de los vectores, y pasarle el otro como parámetro. El vector que se pasa como parámetro no tiene que estar necesariamente representado por un objeto "Vector", sino que también es válida una secuencia de números ("list" o "tuple"). Devuelve el menor ángulo plano entre los dos vectores.
- **Cálculo de vectores perpendiculares**: en ocasiones es necesario calcular el vector perpendicular a uno dado. En el caso de vectores bidimensionales la operación es inmediata (solamente existe una dirección perpendicular, y el sentido se elegirá en función de lo requerido en cada situación): se invierte el orden de sus componentes y se cambia de signo una de ellas (la primera si debe apuntar a la izquierda respecto al sentido del original, la segunda en caso contrario). Si se tratan vectores tridimensionales la complejidad aumenta, ya que existen infinitos vectores perpendiculares a uno dado. Sólo se pueden calcular directamente aquellos que son paralelos a alguno de los planos de referencia, ignorando la componente opuesta a dicho plano y operando como si fueran bidimensionales.
- **Rotación genérica**: en los casos donde los ángulos no sean rectos, o los vectores sean tridimensionales y se busque un resultado no paralelo a un plano de referencia, es necesario usar su método "rotate", que los gira según una matriz de rotación que se le pasa como parámetro. Dicha matriz puede construirse mediante "Matrix.Rotation", siendo necesario importar "Matrix" a priori, que se encuentra también en el módulo "mathutils"; el constructor toma como parámetros el ángulo de rotación (en radianes), el número de dimensiones de la matriz (3 para rotaciones tridimensionales) y el eje alrededor del que se gira (vector o secuencia de números tridimensional). El sentido del eje y el signo del ángulo determinan hacia dónde se gira: si el objeto a rotar (en este caso un vector) se cogiese con la mano derecha, con el dedo pulgar apuntando hacia el eje pasado, un ángulo positivo conllevaría un giro a la derecha, y viceversa.

```
>>> from mathutils import Vector, Matrix
>>> from math import pi

>>> a = Vector([8,0,0])
>>> b = Vector((0,9,2))
>>> c = 2 * (a + b)
>>> c
Vector((16.0, 18.0, 4.0))

>>> b.length
9.219544457292887

>>> a.normalize()
>>> a
Vector((1.0, 0.0, 0.0))

>>> x = Vector([1,0])
>>> x_perp_izquierda = Vector([-x[1],x[0]])
>>> x_perp_derecha = Vector([x[1],-x[0]])
>>> x_perp_izquierda
Vector((-0.0, 1.0))

>>> x_perp_derecha
Vector((0.0, -1.0))

>>> c_perp_izquierda_xz = Vector([-c[2],0,c[0]])
>>> c.angle(c_perp_izquierda_xz)
1.57 # 90°

>>> matriz_rotacion = Matrix.Rotation(pi/2,3,[0,0,1])
>>> matriz_rotacion
Matrix(((0.0, -1.0, 0.0),
        (1.0, 0.0, 0.0),
        (0.0, 0.0, 1.0)))

>>> a.rotate(matriz_rotacion)
>>> a
Vector((0.0,1.0,0.0))
```

Fragmento de código 54. Creación y operaciones con vectores.

7.3. Uso del método “getVectTo”

Este es un método presente en todos los objetos del juego, que se usará con mucha frecuencia dada su gran utilidad. Al llamarlo desde un objeto, pasándole otro objeto o simplemente un punto cualquiera del espacio como parámetro, devuelve tres objetos agrupados en uno de tipo “tuple”: un número real que contiene la distancia entre ellos, un vector unitario que apunta hacia la posición del objeto o punto pasado como parámetro teniendo en cuenta coordenadas globales, y otro vector unitario con la misma información referida a los ejes locales del objeto desde el que se llama.

En el Fragmento de código 55 aparece la salida de una llamada al método desde cualquier módulo que se ejecute en alguno de los ficheros de ejemplo (“demo_control_camara.blend”, etc.), así como la manera de trabajar que se ha seguido a lo largo del proyecto: se asigna directamente una variable (distancia o vector) a uno de los elementos de la secuencia devuelta.

```
from bge import logic

escena_principal = logic.getCurrentScene()
camara_principal = escena_principal.objects['camara_principal']
pelota = escena_principal.objects['pelota']

print(camara_principal.getVectTo(pelota))
# Salida en la consola:
(13.27, Vector((0.98, 0.0, -0.19)), Vector((0.0, 0.0, -1.0)))

distancia = camara_principal.getVectTo(pelota)[0]
vector_camara_pelota = camara_principal.getVectTo(pelota)[1]
vector_camara_pelota_global = camara_principal.getVectTo(pelota)[2]
```

Fragmento de código 55. Uso del método “getVectTo”.

7.4. Manejo de listas y colas

Uno de los tipos de objetos de Python que más se usan en general son las listas (objetos tipo “list”) y sus derivados. En este proyecto en particular se usan con frecuencia, además de listas simples, las colas, que son objetos de tipo “deque” (acrónimo de *double-ended* queue, definido en el módulo “collections”) con una serie de atributos y métodos adicionales que implementan automáticamente el comportamiento “FIFO” (*First In, First Out*).

Un aspecto a tener en cuenta es que las listas no almacenan objetos, sino referencias a ellos. Su comportamiento es similar al de los diccionarios de nombres (“namespace”, ver apartado 3.2.2, página 18), sustituyendo los nombres por índices. Si por ejemplo dos listas contienen una referencia al mismo objeto, o una lista contiene dos referencias al mismo objeto, los cambios en dicho objeto serán visibles en todas las listas desde las cuales se le hace referencia.

A continuación se describen las operaciones que se realizan más a menudo con listas y colas. La mayoría son comunes a ambos tipos, y aquellas que sean específicas de uno de ellos o presenten diferencias se indicarán expresamente. Cada una tiene su correspondencia en el Fragmento de código 56:

- **Creación**: en el caso de las listas lo más común es asignar, a un nombre, un conjunto de elementos entre corchetes, separados por comas. Sin embargo también se puede usar el método estándar, que aparece comentado en el ejemplo: llamar al nombre del tipo (“list”) y pasar como parámetro una secuencia de elementos (entre corchetes, ya que debe ser un solo parámetro en total). Las colas se crean mediante este método estándar, pasando una secuencia de elementos y opcionalmente el parámetro “maxlen”, que indica la capacidad de la cola.
- **Adición de elementos**: se usa el método “append”, que añade por la derecha (el final) el elemento que se pasa como parámetro. Las colas disponen del método “appendleft”, que inserta el elemento en la izquierda (el principio). La capacidad de las listas es infinita, mientras que la de las colas puede limitarse mediante el parámetro “maxlen”;

si una cola está al máximo de su capacidad y se añade un elemento por cualquiera de sus extremos, se eliminará el último elemento presente en el extremo contrario.

- **Acceso y modificación de elementos individuales**: se efectúa escribiendo el nombre de la lista/cola y a continuación el índice del elemento correspondiente entre corchetes. Los índices comienzan por el número 0. Es posible usar índices negativos, en cuyo caso Python suma la longitud de la lista a dicho índice. Si el acceso se hace a la derecha de una asignación, se almacena en el nombre de la izquierda una referencia al objeto al que apunta el elemento de la lista; si se hace a la izquierda, se modifica la referencia almacenada en el elemento de la lista.
- **Acceso y modificación de varios elementos al mismo tiempo (sólo posible en listas)**: se hace mediante *slicing* (cortar en rebanadas). Entre corchetes se indican los puntos de corte (separados por dos puntos ":"), que no representan elementos sino espacios entre ellos: el espacio a la izquierda del primer elemento es el 0, y entre el primero y el segundo es el 1; el número de espacio coincide con el índice del elemento que está a su derecha. Si los puntos de inicio o fin se omiten, se corta por el principio y el final respectivamente. Si el acceso se hace a la derecha de una asignación, la operación devuelve una nueva lista que contiene los elementos entre estos puntos de corte; si se hace a la izquierda, la porción que se encuentra entre los cortes se sustituye por la lista de la derecha, no siendo necesario que el número de elementos coincida. Es posible especificar un parámetro más, el paso (*step*), de manera que sólo se seleccione un elemento de por ejemplo cada dos. Si el paso es negativo se invierte el orden.

  En el ejemplo, en primer lugar se corta entre los espacios 1 y 4, por tanto la lista devuelta contiene los elementos de índices 1,2 y 3. En segundo lugar se omiten los puntos de corte, por tanto se corta por el principio y el final, y se devuelve una lista igual a la original. En la siguiente línea también se omiten los puntos, pero mediante el tercer parámetro se indica que sólo se seleccione un elemento de cada dos. Posteriormente se hace lo mismo, seleccionando todos los elementos y especificando un paso negativo, por lo que se devuelve una lista equivalente pero invertida. Por último se hace que entre los espacios 1 y 2 (en la lista original hay un solo elemento) se coloquen tres elementos, con lo cual la longitud aumentará en dos unidades.
- **Convertir en lista/cola**: en algunas ocasiones se dispone de una secuencia de elementos que no es de tipo "list", o "deque", y se necesita convertirla a dicho tipo para usar sus métodos o realizar otras operaciones que lo requieran. Para ello basta con pasar la secuencia como parámetro al crear una nueva lista. Se debe tener en cuenta que en realidad no se cambia el tipo del objeto, ya que éste es invariable, sino que se crea un nuevo objeto tipo "list" con los mismos elementos que la secuencia pasada como parámetro. En el ejemplo se crea una lista con los elementos de una secuencia tipo "tuple".
- **Copiar lista/cola**: como se explicó en el apartado 3.2.2 (página 18), hacer una asignación del tipo "`nueva_lista = lista_existente`" no ocasiona que se cree una

nueva lista con los mismos elementos que la existente, sino que hace que el nuevo nombre apunte al mismo objeto, y por tanto cualquier manipulación se refleja en ambos. Una copia como se entiende en otros lenguajes (un objeto nuevo con el mismo contenido) se puede conseguir de las siguientes formas:

- Creando una nueva lista (mediante "list") o cola pasando como parámetro la lista/cola existente. En el ejemplo, "lista_2" y "cola_2" se obtienen de esta manera.
- Usando *slicing* para cortar por el principio y el final de manera que la lista (nueva) devuelta contenga todos los elementos de la original. Sólo viable en objetos de tipo "list". La lista "lista_3" del ejemplo se crea de esta forma.
- Usando el método "copy" (no presente en colas) que devuelve una lista nueva con los mismos elementos. En el ejemplo, la lista "lista_4" se crea mediante este método.

Se debe tener en cuenta que, aunque el objeto tipo "list" obtenido sea distinto del original, su contenido es el mismo, es decir, sus elementos apuntan a los mismos objetos. Esto toma relevancia cuando los objetos que componen las listas son mutables (por ejemplo otras listas o diccionarios), en cuyo caso la modificación de uno de ellos se reflejará en todos los elementos de las listas que apunten al mismo, y hay que operar con especial cuidado.

En el código de ejemplo se construye una lista, llamada "lista", que contiene a su vez dos listas de caracteres. Posteriormente se genera una copia y se le da el nombre "lista_2". Aunque son objetos diferentes, su contenido es en ambos casos el mismo conjunto de referencias a dos objetos tipo "list". Al modificar uno de los elementos de estos objetos, aunque las referencias presentes en "lista" y "lista_2" permanezcan invariables, en la práctica el contenido de ambas cambia. A modo de clarificación, en la última línea del bloque se modifica el objeto al que apunta el primer elemento de la lista "lista", que pasa a hacer referencia a otra lista recién creada; este cambio no afecta a "lista_2", cuyos elementos siguen apuntando a las listas anteriores.

```
from collections import deque

lista = ['a','b','c'] # lista = list(['a','b','c'])
cola = deque(['a','b','c'],maxlen = 5)

lista.append('d') # "lista" contiene ['a','b','c','d']
cola.append('d') # "cola" contiene ['a','b','c','d']
cola.appendleft('z') # "cola" contiene ['z','a','b','c','d'] (longitud
máxima)
cola.append('e') # "cola" contiene ['a','b','c','d','e']

ultimo_elemento = lista[-1] # == lista[-1 + 4] == lista[3] == 'd'
cola[0] = 'aa' # "cola" contiene ['aa','b','c','d','e']

slice = lista[1:4] # "slice" contiene ['b','c','d']
slice = lista[:] # "slice" contiene ['a','b','c','d']
slice = lista[::2] # "slice" contiene ['a','c'] (elementos de 2 en 2)
slice = lista[::-1] # "slice" contiene ['d','c','b','a'] (orden
inverso)
lista[1:2] = ['b','bb','bbb'] # "lista" contiene
['a','b','bb','bbb','c','d']

secuencia = (0,1) # "secuencia" es de tipo "tuple"
secuencia_lista = list(secuencia) # "secuencia_lista" es una lista con
los mismos elementos que "secuencia"

lista = ['a','b','c'] # Vuelta a la lista original
lista_2 = list(lista) # Copia de la lista "lista"
cola_2 = deque(cola) # Copia de la cola "cola"
lista_3 = lista[:] # Copia de la lista "lista"
lista_4 = lista.copy() # Copia de la lista "lista"

lista = [['a','b'],['c','d']]
lista_2 = lista[:] # Lista independiente, contiene
[['a','b'],['c','d']]
lista[0][1] = 'f' # "lista" y "lista_2" contienen
[['a','f'],['c','d']]
lista[0] = ['x','y'] # "lista" contiene [['x','y'],['c','d']]
```

Fragmento de código 56. Ejemplos de operaciones con listas y colas.

Estas son las bases del manejo de listas, y aquellas que se utilizan con frecuencia en este proyecto. Para la implementación de operaciones más complejas pueden consultarse las documentaciones de Python y Blender o medios especializados en los mismos. La búsqueda de elementos en listas se trata en el siguiente apartado.

7.5. Búsqueda de elementos en una lista

La necesidad de buscar elementos en listas aparece continuamente al desarrollar algoritmos. Para ello existen dos planteamientos principales: usar el operador “in”, para buscar un elemento equivalente a otro conocido, o recorrer la lista comprobando en cada elemento si alguno de sus atributos cumple la condición que se imponga.

El operador “in”, cuando se usa con la finalidad de hacer una búsqueda, comprueba si un elemento dado, o un conjunto de ellos, están presentes en una lista determinada. La estructura de la comprobación es: `elemento in lista`, y el valor devuelto puede ser “True” o “False”. Este método requiere disponer previamente de un elemento equivalente al buscado, por tanto sólo es adecuado en casos donde éste esté disponible. Estos casos son comúnmente búsquedas

de números en listas o de caracteres en cadenas de caracteres. Un ejemplo usado en este proyecto es la lógica que activa un "truco" al escribir la palabra clave "volar" mediante el teclado: las teclas pulsadas se van almacenando en una lista, y cada vez que se pulsa una nueva se comprueba si en dicha lista está contenida la secuencia de caracteres "volar". En el Fragmento de código 57 se refleja una porción de dicha lógica donde se usa dos veces el operador "in". En la primera se comprueba si la secuencia "volar" está contenida en otra secuencia, la de las teclas que han sido pulsadas hasta el momento. Si sí lo está el operador "in" devuelve "True", por lo que se entra en el bloque "if". Posteriormente se comprueba si la misma secuencia, "volar", está presente en una lista de secuencias, o de cadenas de caracteres, que son los nombres de las propiedades de un objeto del juego. Por último se han incluido algunos ejemplos de uso de este operador y el resultado que producen, para clarificar su funcionamiento.

```
if 'volar' in secuencia_teclas_pulsadas:
    if 'volar' in pelota.getPropertyNames():
        pelota['volar'] = 1 - pelota['volar']
        # Resto de código

print(3 in [1,3,5,7]) # Imprime "True"
print(2 in [1,3,5,7]) # Imprime "False"
print([1,2] in [[1,2],[3,4]]) # Imprime "True"
print([2,3] in [[1,2],[3,4]]) # Imprime "False"
```

Fragmento de código 57. Ejemplo de búsquedas en una lista mediante el operador "in".

En otras situaciones, cuando se requiere encontrar todos los objetos que cumplan una determinada propiedad, o simplemente cuando no es posible usar el operador "in" porque no se conoce el objeto a buscar de antemano, es necesario recorrer las listas completas mediante un bucle y comprobar si cada elemento cumple la condición correspondiente. Un ejemplo es la situación que se da en el Game Engine cuando intervienen varias escenas a la vez y es necesario acceder a objetos de todas ellas desde un mismo *script*. En ese caso se buscan las escenas deseadas en la lista devuelta por la función "logic.getSceneList()". Dado que los elementos de dicha lista son objetos de tipo "KX_Scene", no se puede usar el operador "in" para buscar en ella, ya que para ello sería necesario disponer a priori de un objeto de ese tipo. Una sentencia del tipo "`if 'nombre_escena' in logic.getSceneList():`" sería incorrecta, ya que se estaría buscando un objeto de tipo cadena de caracteres en una lista de objetos de tipo "KX_Scene".

La manera de proceder en general, y en el caso de las escenas en concreto, es la del Fragmento de código 58, donde se supone la existencia de tres escenas, conteniendo cada una al menos un objeto llamado "cubo", y desde un *script* ejecutado desde la escena "escena_A" es necesario acceder a los objetos "cubo" de las tres escenas. En primer lugar se debe obtener una referencia a cada escena, para lo que se recorre la lista de escenas comprobando en cada repetición si el atributo "name" de la escena actual coincide con el de la buscada. Si es así se realizan las acciones correspondientes, en este caso almacenar las referencias a las escenas en nuevas variables. Estas variables permitirán acceder a los objetos buscados de la manera

habitual. Nótese que el código correspondiente a la búsqueda en la lista de escenas es solamente el contenido en el bucle “for”.

```
from bge import logic

escena_A = logic.getCurrentScene()
for escena in logic.getSceneList():
    if escena.name == 'escena_B':
        escena_B = escena
    if escena.name == 'escena_C':
        escena_C = escena

cubo_escena_A = escena_A.objects['cubo']
cubo_escena_B = escena_B.objects['cubo']
cubo_escena_C = escena_C.objects['cubo']
```

Fragmento de código 58. Ejemplo de búsqueda de los objetos de una lista que cumplen una determinada condición.

En el ejemplo anterior se ha tratado un caso específico, donde, al igual que en la gestión de escenas de este proyecto, los elementos de la lista son poco numerosos, y por tanto no es necesario (puede ser incluso contraproducente) introducir lógica para intentar optimizar la búsqueda. En cambio, si se trabaja con listas muy grandes sí es importante hacerlo, ya que el impacto sobre el rendimiento puede ser notable. Las posibles formas de optimizar búsquedas son muchas y la mayoría muy complejas. Además dependen totalmente del contexto en el que se trabaje, por lo que quedan fuera del ámbito de este proyecto. Sin embargo a continuación se explican dos técnicas que es fácil que resulten útiles en el ámbito de Blender y la programación en juegos sencillos.

En primer lugar se expone un ejemplo muy básico donde se busca un único elemento: se tiene una lista de 10000 hipotéticos elementos, donde cada uno de ellos tiene, en su atributo “id”, una cadena de caracteres que sirve de identificador único; si se quiere encontrar el elemento cuyo identificador es el “TNCM”, la manera de proceder sería la que aparece en el Fragmento de código 59. Una vez se ha encontrado el elemento buscado, se coloca la palabra clave “break” para que se deje de ejecutar el bucle “for”. Si el elemento estaba situado en la posición 150, por ejemplo, la introducción de “break” supone que el tiempo que se ha permanecido dentro del bucle representa el 1,5% de lo que se habría tardado sin parar la ejecución.

```
for elemento in lista:
    if elemento.id == 'TNCM':
        elemento_buscado = elemento
        break
```

Fragmento de código 59. Búsqueda de un solo elemento en una lista.

Otra situación simple es aquella donde se busca un número definido de elementos, en cuyo caso se puede usar una variable tipo “int” cuyo valor aumente cada vez que se encuentra un elemento, de manera que cuando el valor de dicha variable es igual al número de elementos buscados se deja de recorrer el bucle. Si por ejemplo en la lista anterior se quisieran

encontrar las entradas con identificador “TNCM” y “LEMD”, una posible manera de operar sería la del Fragmento de código 60:

```
elementos_encontrados = 0
numero_elementos_buscados = 2
for elemento in lista:
    if elemento.id == 'TNCM':
        elemento_TNCM = elemento
        elementos_encontrados += 1
    if elemento.id == 'LEMD':
        elemento_LEMD = elemento
        elementos_encontrados += 1
    if elementos_encontrados == numero_elementos_buscados:
        break
```

Fragmento de código 60. Búsqueda de un número conocido de elementos en una lista.

7.6. Suavizado de movimientos

Cuando un parámetro del juego es controlado continuamente desde un módulo o *script*, en la mayoría de las ocasiones es deseable evitar posibles brusquedades. Por ejemplo, si la velocidad de un objeto es 30, y en un momento dado debe pasar a ser 5, en lugar de aplicar el nuevo valor directamente puede ser preferible alcanzar la nueva velocidad gradualmente. Esto se consigue haciendo una interpolación entre ambos valores.

La estrategia que se ha usado en este proyecto para ello es la siguiente: en lugar de aplicar el valor calculado directamente, se almacenan en una lista de tipo “deque” (los elementos entran por el final, y una vez se alcanza la capacidad máxima se van eliminando los más antiguos), y el valor que realmente se aplica es la media de todos los elementos de dicha lista.

La suavidad dependerá de la longitud de la lista, siendo mayor cuantos más elementos tenga la misma (parámetro “maxlen”). De la longitud también depende el retraso entre el momento en que se produciría un cambio sin usar suavizado y el momento en el que el cambio se percibe cuando sí se usa. En la mayoría de casos habrá que alcanzar un equilibrio entre la suavidad y el retraso.

En el Fragmento de código 61 aparece la estructura más simple de este procedimiento: Como se puede observar, la media se ha calculado de la manera más simple posible; en el proyecto se usa en su lugar la función “calcular_media”, tratada en el apartado 7.9.2 (página 151).

```
from collections import deque

velocidad_deseada = deque(maxlen = 15)

def calcular_y_aplicar_velocidad():
    # Calcular la velocidad que correspondería si no se suavizase

    velocidad_deseada.append(velocidad_calculada)
    velocidad_final = sum(velocidad_deseada) / len(velocidad_deseada)

    # Aplicar velocidad_final
```

Fragmento de código 61. Suavizado de una variable calculando la media aritmética de los valores deseados.

En algunas ocasiones, que aparecerán a lo largo de los diferentes apartados, se ha usado un método diferente para suavizar los cambios: consiste en, para una variable del juego, dado su valor actual y el valor que se desea que tome, aplicar como nueva magnitud el valor actual más una fracción de la diferencia entre ambos. Esta fracción, representada por un número entre 0 y 1, puede definirse como constante al principio del módulo, o calcularse en función de variables del juego, lo que le da una gran flexibilidad.

Sirva como ejemplo el Fragmento de código 62, extraído del módulo que controla el mini-mapa en el juego construido en este proyecto. En él se ajusta el parámetro "ortho_scale" de la cámara, que hace las veces de *zoom*. Como se puede observar, en lugar de aplicar directamente "escala_deseada_camara" se usa un valor intermedio entre el mismo y "escala_actual_camara", controlado por el factor 0.1. En este caso dicho factor se ha introducido directamente, pero podría definirse como constante en el nivel superior si fuera necesario.

```
def controlar_mapa():
    # Resto de lógica

    # Calcular escala_deseada_camara

    escala_nueva_camara =  escala_actual_camara + 0.1*
(escala_deseada_camara - escala_actual_camara);

    camara_mapa.ortho_scale = escala_nueva_camara
```

Fragmento de código 62. Suavizado de una variable aplicando los cambios gradualmente

7.7. Cálculo de la dirección de la pelota

El código que calcula y almacena la dirección de la pelota se ha colocado en el módulo "logica_comun.py" ya que dicha información es usada, además de en el control de la cámara, en el del mini-mapa. Dado que dicho módulo produce información que utiliza el resto, es

importante que al lanzar el juego se ejecute antes que cualquier otro, de manera que las variables necesarias estén creadas antes de ser usadas y no se produzcan errores. Para ello es necesario indicar a Blender que así lo haga activando, en el controlador "Python" que lanza el módulo, la opción correspondiente (ver Figura 17). Por la misma razón, la dirección de la pelota y en general toda la información que se genera en este módulo se almacenan en propiedades de objetos, dado que es necesario acceder a ellas desde diferentes *scripts*.

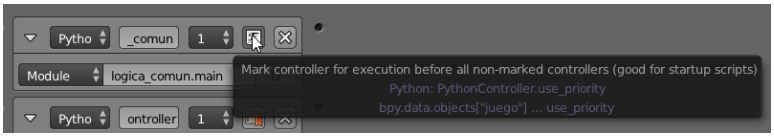


Figura 17. Configuración de un controlador "Python" para que se ejecute antes que el resto (botón situado detrás del cursor)

El proceso consiste básicamente en almacenar en la propiedad "direccion_pelota" del objeto "pelota" el vector bidimensional que resulta de restar de las coordenadas actuales de la pelota las correspondientes a la posición que ocupaba 5 cuadros antes. Para mantener el módulo organizado, el código se ha agrupado en una función, llamada "calcular_direccion_pelota", que es llamada desde "main" en cada cuadro. A continuación se explica cómo funciona y algunos detalles a tener en cuenta.

En primer lugar es necesario disponer de una lista con las últimas posiciones de la pelota. Lo más conveniente es usar un objeto de tipo "deque" (un tipo especial de lista, explicado en el apartado 7.4, página 137), con el que sólo es necesario añadir los elementos nuevos: una vez alcanzada la capacidad máxima, definida por el usuario en el atributo "maxlen", los elementos antiguos se van eliminando conforme se añaden otros nuevos.

Dicha lista se crea e inicializa en el nivel superior, siguiendo lo explicado en el apartado mencionado, y se almacena en la propiedad "historial_posicion_pelota" del objeto "pelota". Para elegir la capacidad de la lista ("maxlen") hay que tener en cuenta todos los usos que se le va a dar. En este caso hay otros cálculos que se basan en la posición de la pelota hasta 10 cuadros antes, con lo cual ese será el valor a utilizar. Se inicializa con la posición de la posición de la pelota; las 10 entradas contendrán el mismo valor, lo que simula que la pelota llevaba unos instantes parada. Para mantener la lista actualizada basta con llamar en cada cuadro al método "append" y pasarle la nueva posición de la pelota. Ver Fragmento de código 63.

En cuanto al cálculo de la dirección propiamente dicho, se desarrolla en su mayoría dentro de la función "calcular_direccion_pelota". Sólo es necesario llamar a esta función en cada cuadro y crear la propiedad "direccion_pelota" en el objeto "pelota" en el nivel superior. Dado que en el primer instante la pelota no lleva ninguna dirección, el valor que se coloca en dicha propiedad es uno cualquiera; en este caso se supone que la pelota avanza hacia adelante (vista desde la cámara), y se coloca el valor correspondiente ([1,0]).

El núcleo de la función "calcular_direccion_pelota" (ver Fragmento de código 63) es su última línea, donde se almacena en la propiedad correspondiente del objeto "pelota" el valor

de la variable local “direccion_pelota”. Este valor puede haberse definido de dos maneras diferentes, dependiendo a grandes rasgos de si la pelota está parada o en movimiento.

En primer lugar (primera línea de la función) se crea, en todos los casos, dicha variable local, de tipo “Vector”; representa a un vector contenido en el plano horizontal, por tanto bidimensional. Su primera dimensión consiste en la diferencia entre la coordenada X de la posición actual de la pelota y la que ocupaba 5 cuadros antes, y su segunda dimensión se calcula de la misma manera considerando la coordenada Y. Nótese que para acceder a los elementos del historial de posiciones se usan índices negativos (el índice aplicado en realidad es la suma del número negativo y la longitud de la lista), de esta forma siempre se tienen en cuenta las cinco últimas posiciones, aunque en un momento dado se decida que el historial debe tener una capacidad diferente.

Posteriormente se comprueba si ha habido movimiento entre las dos posiciones, accediendo al atributo “length” (módulo) del vector recién calculado. Si el módulo es 0 el vector es nulo, lo que indica que la pelota ocupaba la misma posición en los dos cuadros considerados. El vector [0,0] no representa una dirección, y produciría errores en el resto de funciones que requieren la dirección de la pelota.

Se ha decidido por tanto no modificar el valor existente en la propiedad “direccion_pelota” del objeto “pelota” en los casos en que éste se haya movido menos que un cierto umbral. De esta manera en la propiedad siempre se encontrará la última dirección que se ha considerado válida. El discriminar los casos donde el módulo es menor que un umbral en lugar de donde es nulo ayuda en los casos en que la pelota está prácticamente parada: no es deseable que todas las acciones que dependen de cambios de dirección se desencadenen ante cualquier movimiento ínfimo, que en la mayoría de los casos será involuntario. Por ejemplo, considerando que la cámara se orienta automáticamente en la dirección de la pelota, suponiendo que el usuario está observando algo que está ocurriendo en la escena (leyendo un texto sobre una pared, viendo una animación), y teniendo en cuenta que en un juego real probablemente no existiría la opción de parar en seco la pelota, la cámara no dejaría de moverse prácticamente nunca. Es conveniente experimentar con este umbral, asignándole valores extremos para observar el efecto sobre cada función que usa la dirección de la pelota.

En los casos en que el módulo del vector calculado sea mayor que el umbral sí se actualiza el valor de la propiedad “direccion_pelota”, normalizándolo previamente.

En el Fragmento de código 63 aparece toda la lógica necesaria para calcular y almacenar la dirección de la pelota en cada cuadro.

```
from bge import logic
from mathutils import Vector
from collections import deque
from funciones_comunes import reiniciar_cola

escena_principal = logic.getCurrentScene()
pelota = escena_principal.objects['pelota']
pelota['historial_posicion_pelota'] = deque(maxlen=10)
reiniciar_cola(pelota['historial_posicion_pelota'],[pelota.worldPositi
on[0],pelota.worldPosition[1],pelota.worldPosition[2]])
pelota['direccion_pelota'] = Vector([1,0])

def calcular_direccion_pelota():
    direccion_pelota = Vector([pelota['historial_posicion_pelota'][-
1][0] - pelota['historial_posicion_pelota'][-
6][0],pelota['historial_posicion_pelota'][-1][1]-
pelota['historial_posicion_pelota'][-6][1]])
    if direccion_pelota.length > 0.1:
        direccion_pelota.normalize()
        pelota['direccion_pelota'] = direccion_pelota

def main():
    pelota['historial_posicion_pelota'].append([pelota.worldPosition[0
],pelota.worldPosition[1],pelota.worldPosition[2]])
    calcular_direccion_pelota()
```

Fragmento de código 63. Código correspondiente al cálculo de la dirección de la pelota.

7.8. Uso de la matriz “worldOrientation”

Todos los objetos de Blender tienen los atributos, “worldOrientation” y “localOrientation”. Éstos consisten en matrices 3x3 que indican la orientación de los ejes locales del objeto respecto a la de los ejes de referencia. Estos ejes de referencia son los del objeto padre: si no se ha definido ninguna relación, todos los objetos son hijos del objeto escena, y sus matrices “worldOrientation” y “localOrientation” son iguales; si el objeto bajo estudio es hijo de otro objeto, “localOrientation” contendrá la orientación respecto a los ejes de su objeto padre, y “worldOrientation” respecto a los de la escena (globales). En la Figura 18 aparece la estructura de estas matrices, que se explica a continuación.

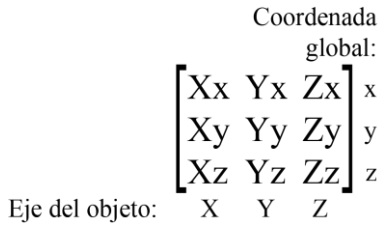


Figura 18. Estructura de las matrices de orientación.

Cada columna corresponde a un eje del objeto, y mediante sus tres elementos representa la dirección y el sentido (respecto a la referencia) en el que apunta ese eje. En la Figura 18 la matriz se ha rellenado con parejas de letras: en cada una, la primera letra indica a qué eje corresponde el valor, y la segunda de qué coordenada de ese eje se trata.

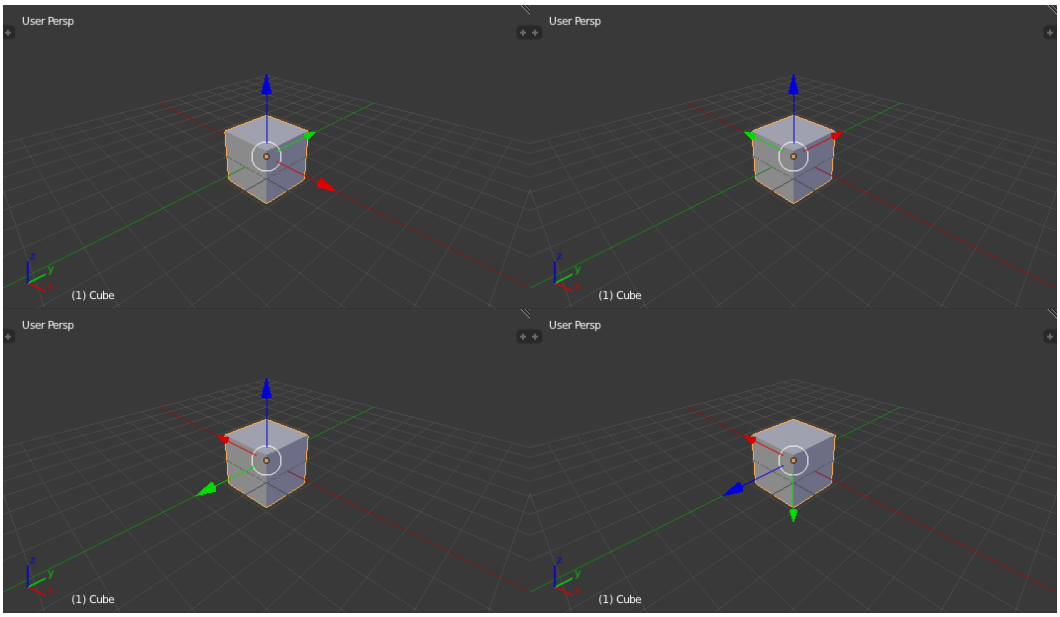
En la Figura 19 se muestran cuatro capturas de pantalla de la escena por defecto de Blender donde, excepto en la primera, el cubo ha sido girado en diferentes direcciones, y debajo de ellas la matriz "worldOrientation" que corresponde a cada una.

En la primera captura el cubo está orientado exactamente igual que los ejes globales (parte inferior izquierda de las capturas), y por tanto su atributo "worldOrientation" es una matriz identidad: la primera columna indica que el eje X del objeto apunta en la dirección global [1,0,0], es decir, en el sentido positivo del eje X, lo cual es coherente con el hecho de que los ejes del objeto coinciden con los globales. Lo mismo sucede con el eje Y, que apunta en la dirección [0,1,0] (eje Y positivo), y con el Z, que lo hace en dirección [0,0,1] (eje Z positivo).

En la segunda captura (superior derecha) se ha girado el cubo 90º alrededor del eje Z, y la matriz "worldOrientation" ha cambiado en consecuencia: como se aprecia en la imagen, ahora el eje X del objeto ya no apunta en la dirección del eje X global, sino en la del eje Y; asimismo, el eje Y ha pasado a apuntar en el sentido negativo del eje X. Estos cambios se reflejan en la matriz, donde la primera columna, correspondiente al eje X del objeto, contiene el vector [0,1,0] (eje Y positivo), y la segunda el vector [-1,0,0] (eje X negativo). Dado que el eje Z sigue apuntando en la misma dirección, la del eje Z global, la columna correspondiente no ha cambiado.

Posteriormente el cubo se ha girado otros 90º alrededor del eje Z, lo que supone una orientación de los ejes X e Y opuesta a la inicial. Esto se refleja claramente en su matriz "worldOrientation", cuyas dos primeras columnas son equivalentes a las del primer caso pero cambiadas de signo. Al ser el giro de nuevo alrededor del eje Z, éste sigue apuntando en el sentido positivo del eje Z global.

Por último, partiendo de la situación anterior, se ha hecho un giro de 90º alrededor del eje X. Esto implica que el eje Y del objeto pasa a apuntar hacia abajo, en el sentido negativo del eje Z, lo que se refleja en la segunda columna de la matriz correspondiente: contiene el vector [0,0,-1] (eje Z negativo). Por su parte el eje Z del objeto deja de apuntar en vertical para hacerlo en el sentido negativo del eje Y, con lo que la tercera columna de la matriz contendrá el vector [0,-1,0] (eje Y negativo). Al haber girado alrededor suyo, el eje X no cambia respecto a la captura anterior.



$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Figura 19. Distintas posiciones de un cubo y las matrices “worldOrientation” correspondientes.

Cuando los giros son de menos de 90º los elementos de la matriz pueden tomar otros valores diferentes de 0 y 1, aunque siempre entre ambos, ya que los vectores que la componen están normalizados.

Si se modifican los valores de la matriz directamente, desde un *script*, para implementar un giro, se debe tener en cuenta que es necesario introducir los valores correctos para todos los ejes que intervienen en dicho giro. Si sólo se modifica un eje, o se introducen valores erróneos, los objetos sufren deformaciones que en la mayoría de los casos son indeseables.

Como ejemplo real se incluye, en el Fragmento de código 64, una porción del módulo que controla el mini-mapa en este proyecto, cuya finalidad es alinear el eje X de un objeto (“icono”) con un vector bidimensional dado (“direccion_deseada_icono”) (se hace así porque, visualmente, el icono es triangular y su pico apunta hacia el sentido positivo del eje X). Este proceso es trivial, ya que “direccion_deseada_icono” está normalizado, con lo cual basta con igualar los dos primeros elementos de la primera columna de “worldOrientation” (eje X) con las dos componentes del vector.

Sin embargo, para que el objeto gire correctamente alrededor del eje Z (en el plano horizontal) es necesario también modificar en la matriz "worldOrientation" los valores correspondientes al eje Y. Para ello hay que tener en cuenta que el eje Y siempre es, en Blender, perpendicular al X y con su sentido positivo apuntando hacia la izquierda respecto al del eje X. Por tanto, en la segunda columna de la matriz "worldOrientation" debe colocarse un vector perpendicular hacia la izquierda a "direccion_deseada_icono", lo que aparece en las dos últimas líneas del Fragmento de código 64.

Destacar que el eje Z del objeto no se modifica porque la rotación se produce solamente en el plano horizontal, y por tanto éste siempre mantiene una orientación fija.

```
# Asignación de variables...

def main():
    # Resto de lógica...
    # Cálculo y normalización de direccion_deseada_icono...

    icono.worldOrientation[0][0] = direccion_deseada_icono[0]
    icono.worldOrientation[1][0] = direccion_deseada_icono[1]
    icono.worldOrientation[0][1] = direccion_deseada_icono[1]
    icono.worldOrientation[1][1] = -direccion_deseada_icono[0]
```

Fragmento de código 64. Modificación manual de la matriz "worldOrientation"

Con el objetivo de ayudar a la comprensión de estas matrices se ha diseñado el fichero de ejemplo "demo_orientation.blend". Consiste simplemente en un cubo que puede rotarse alrededor de los ejes X, Y, y Z mientras se visualizan sus ejes, los de referencia y la matriz "worldOrientation", de manera que se aprecia en tiempo real cómo se ve afectada por los cambios.

Recalcar que las palabras "world" o "local" en el nombre hacen alusión a los ejes que se toman como referencia (los de la escena o los del objeto padre respectivamente). Ambas matrices indican la orientación de los ejes locales del objeto.

7.9. Funciones del módulo "funciones_comunes.py"

A lo largo del proyecto se ha presentado la necesidad de hacer algunas operaciones repetidamente y desde diferentes módulos. Estas operaciones se han convertido en funciones, y para evitar definirlas en cada módulo se ha hecho en uno aparte, llamado "funciones_comunes.py". La manera de usar funciones de módulos externos está explicada en la última parte del apartado 3.3 (página 20). En el caso de este proyecto el módulo se ha incluido en los propios ficheros de ejemplo, por lo que para visualizarlo habrá que seleccionarlo en el editor de texto.

A continuación se explican en detalle las cinco funciones presentes en el módulo: "reiniciar_cola", "calcular_media", "obtener_angulo_x", "obtener_angulo_diferencia" y "coinciden_signos".

Se debe tener en cuenta que estas funciones se han diseñado para su uso en este proyecto, y se ha dado por hecho que se van a utilizar de manera adecuada; por esta razón no se ha implementado ningún tipo de prevención de errores especializada. Por ejemplo, la función "calcular_media" opera directamente con los argumentos pasados, asumiendo que son números; si en su lugar se pasa una cadena de caracteres, el error que aparece en la consola apunta directamente a la línea de la función donde se produce, en vez de indicar la línea de código donde se llamó a la función. Para conseguir esto es necesario considerar cada posible error y explicar su causa, por ejemplo en el caso anterior, tras comprobar que los parámetros pasados no son números: `raise Exception('Los parámetros deben ser números')`.

7.9.1. Inicialización de una lista tipo "deque"

Esta función ("reiniciar_cola") inicializa el objeto de tipo "deque" pasado como primer parámetro, rellenando todas sus posibles posiciones con el valor que se pasa como segundo parámetro. Los objetos tipo "deque" son listas especiales cuyas características se tratan en el apartado 7.4 (página 137).

En esta función se hace uso del atributo "maxlen" (máxima capacidad de la lista) y del hecho de que, al añadir un elemento al final mediante "append", si se ha alcanzado la capacidad máxima se elimina el que ocupa el primer lugar.

Dado que, independientemente del estado de la lista pasada, el objetivo es hacer que todas sus posibles posiciones contengan el mismo valor (es decir, apunten al mismo objeto), el código consiste en un bucle "for" que se ejecuta un número de veces igual al atributo "maxlen", dentro del cual se llama al método "append" de la lista pasando como parámetro el valor correspondiente. Si estaba vacía simplemente se llenará, y de lo contrario en cada ciclo se desechará uno de los valores que tenía anteriormente. En el Fragmento de código 65 aparece el código correspondiente.

```
def reiniciar_cola(cola, valor):
    for i in range(cola.maxlen):
        cola.append(valor)
```

Fragmento de código 65. Inicialización de una lista tipo "deque".

7.9.2. Cálculo de la media aritmética de los valores de una lista

Dado que en numerosas ocasiones a lo largo del proyecto es necesario calcular la media aritmética de un conjunto de valores, el proceso de cálculo se ha agrupado en una función ("calcular_media"), que se ha colocado en el fichero "funciones_comunes.py", de manera que se pueda importar y usar cuando sea necesario.

Se ha implementado de forma que es posible también pasar como parámetro un conjunto de conjuntos de números (ya sean listas, vectores, tuplas o cualquier derivado), y calcular la media de todos los enésimos elementos de dichos conjuntos. Para ello la función debe

distinguir si los elementos de la lista pasada son números (tipo "float" o "int") o no, y en este último caso es necesario pasar como parámetro un número indicando de qué dimensión de las listas se debe calcular la media. Esto sería útil por ejemplo si se tuviera una lista con las 20 últimas posiciones de una cámara (vectores tridimensionales), y hubiese que calcular la altura media; la llamada a la función sería como la siguiente: `media = calcular_media(lista_posiciones,2)`.

En el Fragmento de código 66 aparece el código completo. La función "isinstance" devuelve "True" si el primer parámetro pasado es del tipo que se indica mediante el segundo, y "False" en caso contrario; por ejemplo, `isinstance(1,int)` devuelve "True".

```
def calcular_media(lista,dimension = 0):
    total = 0
    if isinstance(lista[0],int) or isinstance(lista[0],float):
        total = sum(lista)
    else:
        for i in range(len(lista)):
            total += lista[i][dimension]
    return total/len(lista) if len(lista) > 0 else 0
```

Fragmento de código 66. Código de la función "calcular_media".

7.9.3. Cálculo del ángulo que forma un vector del plano horizontal con el eje X

Dado que este cálculo se realiza en diferentes módulos, los pasos necesarios se han agrupado dentro de la función "obtener_angulo_X", definida en el módulo "funciones_comunes.py", guardado en un archivo de texto externo al fichero de Blender. Su código aparece en el Fragmento de código 67.

Esta función toma como parámetro un vector bidimensional y devuelve el ángulo, en radianes, que forma con el eje X ([1,0]). Su funcionamiento se basa en el método "angle" de la clase "Vector" (devuelve el ángulo en valor absoluto, entre 0 y pi radianes, que forma el vector desde el que se llama con el vector que se pasa como parámetro). El valor devuelto por este método debe ser ajustado por dos razones:

- El método "angle" no toma un vector como referencia y calcula el ángulo que otro forma con él, sino que devuelve el menor ángulo absoluto entre ellos (siempre entre 0 y pi rad). En la mayoría de ocasiones es necesario saber si el vector forma un ángulo positivo o negativo con el eje X, lo que se consigue de la siguiente manera: dado que la referencia siempre es el eje X ([1,0]), basta con fijarse en la segunda componente del vector; si es menor que 0 rad, el ángulo real que forma el vector con el eje X es el devuelto por "angle" cambiado de signo.
- De la corrección anterior resulta un ángulo entre –pi y pi rad. Por comodidad, se prefiere que esté representado entre 0 y 2*pi rad, para lo cual basta con sumar 2*pi radianes a los ángulos menores que 0.

El vector que se pasa como parámetro a la función puede ser de clase "Vector" o simplemente una lista de dos números, ya que el método "angle" admite ambos.

```
def obtener_angulo_x(vector):
    angulo = Vector([1,0]).angle(vector)
    if vector[1] < 0:
        angulo = -angulo + 2*pi
    return angulo
```

Fragmento de código 67. Código de la función "obtener_angulo_x".

7.9.4. Cálculo del ángulo diferencia entre dos dados

En varias ocasiones es necesario calcular el ángulo diferencia entre otros dos, tomando uno de ellos como referencia, de forma que el ángulo que el otro forma con él pueda resultar positivo o negativo. Para ello se usará la función "obtener_angulo_diferencia", definida en el módulo "funciones_comunes.py" y cuyo código aparece en el Fragmento de código 68.

En esta función toma dos ángulos expresados en radianes como parámetros ("angulo_2" y "angulo_1"), siendo "angulo_1" el ángulo que se toma como referencia. En ella se realiza la operación: `angulo_2 - angulo_1`, teniendo en cuenta que su valor debe situarse entre –pi y pi radianes: si el ángulo "angulo_2" es de 350º, y el "angulo_1" de 10º, la diferencia debe ser de -20º, no de 350-10=340º (en los ejemplos los ángulos se expresan en grados por considerarse más intuitivos). Para cumplir esta condición, la diferencia nunca debe ser mayor que pi ni menor que –pi radianes; en el primer caso al valor obtenido se le deben restar 2*pi radianes, en el último se le debe sumar dicha cantidad. La razón para esto es que siempre debe buscarse el camino más corto para llegar desde el ángulo dado hasta el de referencia, y este camino nunca es de más de 180º: si la diferencia resultara de 190º, lo apropiado sería girar 170º en sentido contrario. Si no se hiciera esta corrección los resultados serían erróneos cuando cada ángulo estuviera en un lado distinto del eje X.

```
def obtener_angulo_diferencia(angulo_2,angulo_1):
    diferencia = angulo_2 - angulo_1
        if diferencia > pi:
            diferencia -= 2*pi
        if diferencia < -pi:
            diferencia += 2*pi
    return diferencia
```

Fragmento de código 68. Código de la función "obtener_angulo_diferencia".

7.9.5. Comprobación de la igualdad de signos entre dos números

Aunque se trata de una operación muy simple, que en la mayoría de los casos puede resumirse en una línea de código, al aparecer varias veces a lo largo del proyecto y ser dicha línea bastante extensa (con lo que perjudica la legibilidad) se ha convertido en una función.

Esta función, llamada “coinciden_signos”, toma como parámetros dos números (aquellos cuyos signos se deben comparar) y devuelve 1 si sus signos coinciden y 0 en caso contrario.

La igualdad de signos entre los números se determina comprobando si la suma de sus valores absolutos es igual al valor absoluto de la suma. En caso negativo los signos son distintos, mientras que en caso afirmativo puede que o bien los signos sean iguales o bien uno de los números sea el cero. Para discernir a qué se debe un resultado afirmativo, se hace que la función devuelva cero directamente si alguno de los parámetros pasados es cero.

En el Fragmento de código 69 se incluye el código correspondiente. En lugar de una estructura if/else tradicional se ha hecho uso de una expresión condicional (u operador ternario), que es más compacta y práctica si la toma de decisiones es simple.

```
def coinciden_signos(valor_1,valor_2):
    if valor_1 == 0 or valor_2 == 0:
        return 0
    return 1 if fabs(valor_1 + valor_2) == fabs(valor_1) +
fabs(valor_2) else 0
```

Fragmento de código 69. Comprobación de la igualdad de signos entre dos números.

8. CONCLUSIONES

Con este proyecto se ha desarrollado una guía introductoria a uno de los aspectos más complejos y especializados de Blender, que es el control de su motor de videojuegos mediante programas escritos en Python.

Se ha organizado en una pequeña parte descriptiva, centrada en el lenguaje Python y en las bases de su uso para programar el motor de videojuegos (Game Engine) de Blender, y otra práctica, que constituye la mayoría del proyecto, donde se estudian de manera progresiva ejemplos concretos de uso del mismo.

En la sección descriptiva se ha tratado tanto el funcionamiento más básico del lenguaje Python, especialmente las características que difieren de otros lenguajes de programación tradicionales, como su relación con Blender en particular, explicando las diferentes partes de la API de Blender para Python, y las posibles estrategias de uso.

La parte práctica, dado que esta interacción entre Blender y Python ofrece un rango de posibilidades muy amplio, se ha preferido centrarla en tres áreas concretas e investigarlas a fondo: el control del objeto protagonista, de la cámara y del mapa de orientación.

Por una parte, estos tres aspectos se han explicado exhaustivamente, y se han llevado hasta un nivel relativamente alto. Asimismo se han intentado minimizar las dependencias, tanto entre ellos como con la escena que se ha usado como ejemplo, de manera que sea sencillo usar los programas generados en otras aplicaciones.

Por otra, la mayoría de los problemas que ha sido necesario resolver durante el desarrollo no son específicos de ninguna de las tres áreas, sino que son de carácter general, por lo que sus explicaciones podrán usarse al afrontar otras situaciones.

En el apartado personal, la realización de este proyecto ha constituido un gran beneficio: por un lado se han ampliado en gran medida los conocimientos sobre Blender y se han estudiado los fundamentos de Python, y por otro se han mejorado la autodisciplina y la constancia en el trabajo con objetivos a medio plazo.

Las mayores dificultades se han encontrado, en particular, al diseñar el sistema que evita los choques de la cámara en cualquier dirección y, en general, al tener que poner fin a la etapa de desarrollo para abordar la redacción de la memoria, ya que por una parte se han tenido que dejar de lado aspectos interesantes que no estaban suficientemente desarrollados, y por otra no se han podido seguir mejorando todo lo que se habría querido los que sí se han estudiado.

Como punto mejorable puede decirse que probablemente no se haya conseguido el balance correcto entre el número de temas tratados y nivel de detalle ya que, aunque se han estudiado en profundidad tres de ellos, no ha sido posible abordar alguno de los que inicialmente se habían propuesto.

A continuación se comentan algunas mejoras concretas que podrían hacerse directamente sobre los módulos, partiendo del estado en que aparecen en esta memoria y en los ficheros de ejemplo:

- Control del objeto protagonista:
    - Refinar el criterio usado para decidir si el salto debe permitirse o no, para evitar que se haga cuando, aunque se detecte contacto, este se deba a una superficie completamente vertical.
- Control de la cámara:
    - Cambiar su comportamiento cuando choca contra una pared; en lugar de parar la velocidad, hacer que sea tangente a la pared, de manera que se acerque al objeto protagonista.
    - Modificar el mecanismo que aplica velocidad lateral para evitar obstáculos para que se haga también en vertical.
- En general:
    - Implementar un sistema de logros, de forma que el juego tenga objetivos, y existan maneras de favorecer o perjudicar su consecución. Una versión básica podría implementarse simplemente con el uso de sensores (el protagonista pasa por un lugar, hace contacto con o está cerca de un objeto) y el almacenamiento de información (almacenar cuántas veces se ha producido un evento, la "vida" del protagonista, el "dinero"), ambos cubiertos ampliamente en este proyecto.
    - Añadir un inventario, o un sistema de información, que permita saber el estado de diferentes parámetros, como los relacionados con el sistema de logros anterior. Podría conseguirse superponiendo una escena, de la misma forma que en el caso del mini-mapa de orientación, en la que aparezcan objetos (a modo de iconos, o texto) que representen los datos correspondientes.

Al margen de lo anterior, de entre la multitud de opciones que presenta el uso combinado de Blender y Python se sugieren algunas cuyo estudio en futuros proyectos o trabajos constituiría una continuación lógica de éste:

- Estudio extensivo del funcionamiento de las escenas y su uso mediante la API de Python. En este proyecto sólo se ha explicado la posibilidad más básica (superponer una escena al principio de la ejecución), sin embargo diferentes pruebas que se han hecho durante el desarrollo indican que la complejidad que presentan operaciones como saltar entre escenas o reiniciarlas, las relaciones entre ellas y el tratamiento de las variables que se generan en cada una, es alta y merece un estudio aparte. Éste incluiría elementos comunes en los videojuegos, como la presencia de diferentes niveles, o la posibilidad de pausar, reiniciar o finalizar el juego.
- Carga selectiva del escenario. Es un aspecto que puede ser conveniente desde el punto de vista de los objetivos del juego (no mostrar parte del nivel hasta que se hayan conseguido determinados logros) y que en aplicaciones grandes es necesario por razones de rendimiento: en general no es práctico cargar elementos lejanos, con los que probablemente no se podrá interactuar desde la distancia, sino que sólo se

renderizan aquellos que están en un radio determinado, y probablemente dentro o cerca del campo de visión de la cámara, para hacer un uso eficiente de los recursos.

- Uso de sonido. Un elemento prácticamente imprescindible en cualquier juego, para el que Blender ofrece una gran cantidad de posibilidades a través de la API de Python, incluyendo el control de la posición en el espacio tridimensional de la fuente, la ganancia o el “pitch” del sonido. Estas posibilidades podrían usarse por otra parte para desarrollar aplicaciones interactivas que demostrasen algunos conceptos que se estudian en asignaturas impartidas en la Escuela, tales como el efecto Doppler o la localización psicoacústica de sucesos auditivos.

BIBLIOGRAFÍA Y MATERIAL DE CONSULTA

- Aforismos de Python (The Zen of Python): http://www.python.org/dev/peps/pep-0020/.
- Documentación oficial de Python: http://docs.python.org.
- Colección de tutoriales sobre Python, adaptados a diferentes niveles: http://wiki.python.org/moin/BeginnersGuide/Programmers/.
- Documentación oficial de la API de Blender para Python (última versión disponible): http://www.blender.org/documentation/blender_python_api_2_68a_release/.
- Sitio web de intercambio de conocimientos sobre programación (Stack Overflow): http://stackoverflow.com/.
- Sitio web de intercambio de conocimientos sobre Blender (Blender Artists): http://www.blenderartists.org/.
- Sitio web con una gran cantidad de artículos de calidad sobre Python: http://effbot.org/. Entre ellos, de lectura especialmente recomendada para complementar este proyecto:
    - Sobre objetos en Python: http://www.effbot.org/zone/python-objects.htm.
    - Sobre listas en Python: http://www.effbot.org/zone/python-list.htm.
- Proyecto Fin de Carrera de Joaquín Riezu González: “Entendiendo el Game Engine”, junio 2011.

ANEXO I

A continuación se presenta el código, completo y comentado, del *script* que realiza una copia de la escena para ser utilizada en el mini-mapa, llamado “crear_mapa.py”.

```
# Importar los módulos necesarios
import bpy
from math import pi, radians

# Definir como variables los nombres concretos para hacer más fácil su modificación
a posteriori, si ésta fuera necesaria
nombre_escena_principal = 'escena_principal'
nombre_objeto_protagonista = 'pelota'

# Obtener, de la lista de escenas, una referencia a la principal
escena_principal = bpy.data.scenes[nombre_escena_principal]

# Obtener, de la lista de objetos de la escena principal, una referencia al
protagonista
objeto_protagonista = escena_principal.objects[nombre_objeto_protagonista]

# Definir la altura relativa a la que se colocará la copia de la escena
altura_mapa = -1000

def crear_material(color_material):
    # Construir el nombre del material correspondiente al color, convirtiendo el
vector pasado en una cadena de caracteres
    nombre_material = 'material_mapa_' + str(color_material)
    # Dar un valor por defecto a la variable que almacena si el material ya existe.
En ese caso se devolverá el existente en lugar de crear uno nuevo
    material_ya_existe = False
    # Recorrer la lista de materiales asociados a un objeto
    for material in bpy.data.materials:
        # Comprobar si el nombre del material actual coincide con el construido
anteriormente
        if material.name == nombre_material:
            # Si coinciden, el material ya existe, se da valor "True" a la variable
correspondiente
            material_ya_existe = True
            # Al salir del bucle, la variable "material" mantiene la referencia al
objeto actual
            break
    # Si el material no existía previamente, hay que crearlo
    if material_ya_existe is False:
        # Crear un nuevo material con el nombre correspondiente
        material = bpy.data.materials.new(nombre_material)
        # Asignar su color al pasado como parámetro
        material.diffuse_color = color_material
        # Configurar su parámetro "Intensity"
        material.diffuse_intensity = 1.0
        # Configurar su parámetro "Ambient"
        material.ambient = 0
        # Configurar su parámetro "Emit"
        material.emit = 0.2
        # Activar la opción "Shadeless"
        material.use_shadeless = True
    # Devolver una referencia bien al material recién creado o bien al existente
    return material

def main():

    # Deseleccionar los objetos que pudieran estar seleccionados para que no sean
eliminados
    bpy.ops.object.select_all(action='DESELECT')
    # Recorrer la lista de objetos de la escena principal
```

```
    for objeto in escena_principal.objects:
        # Comprobar si el nombre del objeto termina en "_mapa"
        if '_' in objeto.name and objeto.name.split('_')[-1] == 'mapa':
            # Hacer activo el objeto actual
            escena_principal.objects.active = objeto
            # Seleccionar el objeto actual (objeto activo != objeto/s
seleccionado/s)
            objeto.select = True
            # Almacenar en una variable una referencia a la estructura interna
(mesh) del objeto activo (el actual)
            # Las estructuras deben tener 0 usuarios para poder ser borradas, por
lo que se debe eliminar primero el objeto
            estructura = objeto.data
            # Borrar el objeto
            bpy.ops.object.delete()
            # Borrar la estructura. Se encierra dentro de una estructura try/except
porque algunos objetos podrían no tener estructura (empties, si los hubiera), y de
otra forma se produciría un error
            try: # Empties no tienen estructura
                bpy.data.meshes.remove(estructura)
            except:
                pass

    # Recorrer la lista de objetos de la escena principal
    for objeto_original in escena_principal.objects:
        # Comprobar si el objeto tiene asociado algún material
        # Si el objeto no tiene material, se entiende que no hay que representarlo
(será una lámpara, cámara, etc)
        if hasattr(objeto_original.data,'materials'):
            # Crear un nuevo objeto, de nombre el del original terminado en "_mapa"
y de estructura una copia de la original
            objeto_duplicado = bpy.data.objects.new(objeto_original.name +
'_mapa',objeto_original.data.copy())
            # Asignar el objeto recién creado a la escena principal
            escena_principal.objects.link(objeto_duplicado)
            # Igualar la rotación del nuevo objeto a la del original
            objeto_duplicado.rotation_euler = objeto_original.rotation_euler
            # Igualar la escala del nuevo objeto a la del original
            objeto_duplicado.scale = objeto_original.scale
            # Definir la nueva posición como la del objeto original, sumando el
desvío vertical a la coordenada Z
            nueva_localizacion = objeto_original.location.copy() # Si no se usa
"copy()", en el siguiente paso se modificaría el objeto original
            nueva_localizacion[2] = nueva_localizacion[2] + altura_mapa
            # Asignar la posición calculada al nuevo objeto
            objeto_duplicado.location = nueva_localizacion

            # Comprobar el tipo físico del objeto original, y su nombre.
            if objeto_original.game.physics_type != 'STATIC' and
objeto_original.name != nombre_objeto_protagonista:
                # Hacer activo el objeto nuevo
                escena_principal.objects.active = objeto_duplicado
                # Añadir una propiedad al objeto activo, de nombre
"objeto_dinamico" y tipo número entero
                bpy.ops.object.game_property_new(type='INT',name='objeto_dinamico')
                # Dar el valor 1 a la propiedad recién creada
                objeto_duplicado.game.properties['objeto_dinamico'].value = 1

            # Recorrer la lista de materiales del objeto nuevo
            for i in range(len(objeto_duplicado.data.materials)):
                # Eliminar el último elemento de la lista
                objeto_duplicado.data.materials.pop(0,True)
            # Crear una lista con los valores RGB (0 - 1) del color "diffuse" del
primer material presente en la lista de materiales del objeto original, redondeados
a un decimal
            color_material =
[round(objeto_original.data.materials[0].diffuse_color[0],1),round(objeto_original.
```

```
data.materials[0].diffuse_color[1],1),round(objeto_original.data.materials[0].diffu
se_color[2],1)]
            # Crear un nuevo material de dicho color
            material_nuevo = crear_material(color_material)
            # Colocar el material recien creado en la lista de materiales del
objeto nuevo
            objeto_duplicado.data.materials.append(material_nuevo)

    # Crear una cámara, en la posición y con la rotación pasadas como parámetros
    bpy.ops.object.camera_add(location=[0,0,altura_mapa+35],rotation=[0,0,radians(-
90)])
    # Almacenar una referencia a ella en una variable (al ser creada se convierte
en el objeto activo automáticamente)
    camara_superior = bpy.context.active_object
    # Modificar su nombre
    camara_superior.name = 'camara_mapa'
    # Configurar el tipo de lente
    camara_superior.data.type = 'ORTHO'
    # Configurar el parámetro "Ortho Scale" (similar al zoom)
    camara_superior.data.ortho_scale = 30

    # Obtener una referencia al objeto recién creado que representa al protagonista
(su nombre debe ser el mismo, terminado en "_mapa"
    icono = bpy.data.objects[nombre_objeto_protagonista + '_mapa']
    # Almacenar su posición
    localizacion_icono = icono.location
    # Deseleccionar todos los objetos
    bpy.ops.object.select_all(action='DESELECT')
    # Hacer activo el objeto que representa al protagonista
    escena_principal.objects.active = icono
    # Seleccionar el objeto que representa al protagonista
    icono.select = True
    # Borrar el objeto seleccionado
    bpy.ops.object.delete()

    # Crear un cono de base triangular, en la posición almacenada anteriormente y
tumbado sobre el plano horizontal

bpy.ops.mesh.primitive_cone_add(vertices=3,location=localizacion_icono,rotation=[ra
dians(90),radians(0),radians(90)])
    # Aplicar la rotación
    bpy.ops.object.transform_apply(rotation=True)
    # Almacenar una referencia al cono (al ser creado se convierte en el objeto
activo)
    icono = bpy.context.active_object
    # Modificar su nombre
    icono.name = 'icono_mapa'
    # Establecer su altura sobre el resto de la copia
    icono.location[2] = altura_mapa + 15
    # Configurarlo como "Ghost", para que no interactúe con otros objetos
    icono.game.use_ghost = True
    # Asignarle un material de color azul
    icono.data.materials.append(crear_material([0,0,1]))

    #Limpiar la seleccion
    bpy.ops.object.select_all(action='DESELECT')

main()
```